University of Bath

PHD

Integrated-Key Cryptographic Hash Functions

Al-Kuwari, Saif

Award date:
2011

Awarding institution:
University of Bath

Link to publication

Alternative formats
If you require this document in an alternative format, please contact:
openaccess@bath.ac.uk

Copyright of this thesis rests with the author. Access is subject to the above licence, if given. If no licence is specified above, original content in this thesis is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC-ND 4.0) Licence (https://creativecommons.org/licenses/by-nc-nd/4.0/). Any third-party copyright material present remains the property of its respective owner(s) and is licensed under its existing terms.

Take down policy
If you consider content within Bath's Research Portal to be in breach of UK law, please contact: openaccess@bath.ac.uk with the details. Your claim will be investigated and, where appropriate, the item will be removed from public view as soon as possible.

Download date: 06. May. 2023

Integrated-Key Cryptographic Hash Functions

submitted by

Saif Mohammed S. A. Al-Kuwari

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Computer Science

September 2011

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with its author. A copy of this thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with the author and they must not copy it or use material from it except as permitted by law or with the consent of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signature of Author .....

Saif Mohammed S. A. Al-Kuwari

و ما توفيقي إلا بالله

"I cannot succeed except through God"

Abstract

Cryptographic hash functions have always played a major role in most cryptographic applications. Traditionally, hash functions were designed in the keyless setting, where a hash function accepts a variable-length message and returns a fixed-length fingerprint. Unfortunately, over the years, significant weaknesses were reported on instances of some popular "keyless" hash functions. This has motivated the research community to start considering the dedicated-key setting, where a hash function is publicly keyed. In this approach, families of hash functions are constructed such that the individual members are indexed by different publicly-known keys. This has, evidently, also allowed for more rigorous security arguments. However, it turns out that converting an existing keyless hash function into a dedicated-key one is usually non-trivial since the underlying keyless compression function of the keyless hash function does not normally accommodate the extra key input. In this thesis we define and formalise a flexible approach to solve this problem. Hash functions adopting our approach are said to be constructed in the integrated-key setting, where keyless hash functions are seamlessly and transparently transformed into keyed variants by introducing an extra component accompanying the (still keyless) compression function to handle the key input separately outside the compression function. We also propose several integrated-key constructions and prove that they are collision resistant, pre-image resistant, 2nd pre-image resistant, indifferentiable from Random Oracle ($\mathcal{RO}$), indistinguishable from Pseudorandom Functions (PRFs) and Unforgeable when instantiated as Message Authentication Codes (MACs) in the private key setting. We further prove that hash functions constructed in the integrated-key setting are indistinguishable from their variants in the conventional dedicated-key setting, which implies that proofs from the dedicated-key setting can be naturally reduced to the integrated-key setting.

Acknowledgement

First and foremost, all thanks and praised are due to God, for giving me the strength and patience to complete this thesis, like many other things in life. This project (and everything I have ever done) could not have been completed without his blessings.

On Earth, I would like to thank my supervisors, Prof. James H. Davenport and Dr. Russell J. Bradford, for their support and supervision throughout my PhD. I was very fortunate to have been supervised by them, I would not have asked for better supervisors. James has always provided very insightful comments, which not only improved my thesis, but also even improved my understanding of my own work. I appreciate the endless emails we exchanged, even during weekends and his holidays. This thesis would not have been possible without all the time and effort he put in supervising it. I would also like to thank my examiners, Dr. Liqun Chen and Prof. Guy McCusker for their valuable comments on my thesis. Guy is also our director of studies so I would like to reiterate my gratitude to him, this time for his support as a director of research, especially in the first two years of my PhD.

During the time I spent in Bath, I had the privilege to meet many great people. Unfortunately, this acknowledgement is too small to list them all, but I would like to particularly thank Dalia Khader, Anupriya Balikai, Ana Calderon, Martin Brain, Fabio Nemetz and Mayuree Srikulwong for their friendship and support through the years.

From the university of Essex (where I did my undergraduate studies), I would like to thank Prof. Peter Higgins and Dr. Alexei Vernitski, who introduced me to the world of cryptography. When I first attended my first lecture in cryptography given by Alexei, I immediately knew that this topic (whose name I could not even pronounce correctly) would be an important part of my life. My interest in cryptography has further been boosted the following year when I opted to take a mathematically oriented cryptography course given by Prof. Higgins. I am also in dept to Dr. David Hunter who was my final year project supervisor. David taught me that research is a passion more than anything else, the more I do research the more I appreciate how right he was.

Thanks to my family for always being there for me. Thanks to my mother for putting up with me being away for a long time, and for my father for being such a great role model. Thanks for the endless prayers of my grandmother and aunts, and for my lovely nieces and nephews who were born while I was abroad. I would also like to thank Alice and Bob for their great company in my long journey through the world of cryptography, they are truly my and every cryptographer's heroes.

Finally, thanks to everyone who believed in me for giving me the opportunity to earn their trust, and for everyone who did not believe in me for giving me the opportunity to prove them wrong.

Contents

List of Figures

List of Symbols

f	keyless compression function (variants f_x, f_y, f_c)
h	keyed compression function (synonym to f_K)
g	keyed integration function (variants g_x, g_y, g_c, g_1, g_2)
m	length of a single message block
n	length of the chaining variable and final hash value
l	length of total input to a compression function ($l = m + n$)
H	keyless hash function (denoted H_K when keyed)
C	dedicated-key hash function
$\hat{C}$	integrated-key hash function
$\mathcal{M}$	message space (usually bounded by, e.g., $\{0,1\}^{64}$)
$\mathcal{K}$	key space ($k_i \in \mathcal{K}$ index members of a hash function family)
$\mathcal{F}$	random oracle (an ideal primitive)
$\mathcal{H}$	ideal compression function (variants $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$)
$\mathcal{G}$	ideal integration function (variants $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$)
S	compression function simulator (variants S_1, S_2, S_3)
R	integration function simulator (variants R_1, R_2, R_3)
A	adversary (usually) attacking a hashing construction
B	adversary (usually) attacking a compression function
G	adversary attacking an integration function
D	distinguisher adversary (used in chapters 4, 5 and 6)
q_A	number of queries sent by an adversary (or distinguisher) A
ϵ	the success advantage of a particular game
t_A	the running time of an adversary A (t_i is a tag in chapter 5)
ℓ	the number of blocks in a message
$e(L)$	the number of blocks in a message of length L-bit
ℓ_A	the number of blocks in all queries sent by adversary A
F	dummy relay algorithm (variants are F_1, F_2)
τ_x	cost of calling function x
T	tag (produced by a MAC algorithm; used in chapter 6)

List of Notation

$x \xleftarrow{\$} \mathcal{X}$	choosing a value uniformly at random from the set $\mathcal{X}$
$y \xleftarrow{\$} A(x)$	adversary A outputs y on input x
$\{0,1\}^n$	all binary strings of length n
$\{0,1\}^*$	all binary strings
$\perp$	the empty string
$\wedge$	logical operator AND
$\vee$	logical operator OR
$\star$	arbitrary composition operation
$\circ$	full composition (definition 6.2.2)
$\triangle$	partial composition (definition 6.2.3)
$\rightarrow$	forward query (generated by a distinguisher)
$\leftarrow$	inverse query (generated by a distinguisher)
$\bowtie$	indistinguishability (definition 6.1.1)
$x\|\|y$	the concatenation of x and y
x, y	shorthand for $x\|\|y$
$\|M\|$	length of bit string M
$\langle M \rangle_\beta$	β-bit encoding of the length of bit string M
$M[i]$	the i-th block of message M
$M[i \dots j]$	concatenations of the i-th up to the j-th blocks of message M
M^i	sequence of message blocks $M_1, M_2, \dots, M_i$
$[M]^n$	most significant n-bits of bit string M
$[M]_n$	least significant n-bits of bit string M
$M^{(i)}$	the input to the i-th application of a compression function
X^Y	component X has oracle access to component Y
$\lceil x \rceil$	the celling function
$Dom(F)$	the domain of function F
$Rng(F)$	the range of function F
$\Pr[X\|Y]$	the probability that X will occur given Y occurred
$\mathrm{P}^F_{X\|Y}$	the probability that system F will output Y given X
$\mathbf{Adv}^{\text{xxx}}_A$	the advantage of adversary A on game xxx
$(\mathcal{X} \rightarrow \mathcal{Y})$-$\mathbf{R}$	System $\mathbf{R}$ with input space $\mathcal{X}$ and output space $\mathcal{Y}$

List of Abbreviations

a-xxx	always-xxx, where xxx $\in$ {Sec,Pre}
Adv	Advantage
c-iMD*	Output-based iMD
Coll	Collision
CR	Collision Resistance
Dom	Domain (of a function)
e-xxx	everywhere-xxx, where xxx $\in$ {Sec,Pre}
GRS*	Generalised Random System
ICM	Ideal Cipher Model
iMD*	integrated-key Merkle-Damgård
*inv**	Full inversion
IV	Initialisation Vector
MAC	Message Authentication Code
MPP	Multi-Property-Preserving
*p-inv**	Partial inversion
NIST	National Institute of Standards and Technology
PPC	Prepend-Permute-Chop
Pre	Pre-image
PRF	Pseudorandom Function
PRO	Pseudorandom Oracle
Rng	Range (of a function)
$\mathcal{RO}$	Random Oracle
ROM	Random Oracle Model
s-Sec*	Fixed Suffix 2nd Pre-image
Sec	Second pre-image
SIMD	Single Instruction, Multiple Data
TCR	Target Collision Resistance
UOWHF	Universal One Way Hash Function
x-iMD*	Input-based iMD
y-iMD*	chaining-based iMD

*novel

List of Definitions, Theorems, Lemmas and Corollaries

Chapter 1

Introduction

Cryptography is the science of secret communication, and along with cryptanalysis (the science of breaking cryptographic schemes), it constitutes what is known as *cryptology*. Cryptography has a long history [90] dating back to the A.D. era where it was first used when humans realised the importance of information secrecy, confidentiality and authenticity. Motivated mainly by political interests, cryptography played a major role in preserving the secrecy of classified information. Generally, cryptography can be categorised as either symmetric or asymmetric. In symmetric cryptography, both the sender and the receiver share a secret key which they use to encrypt/decrypt the (sensitive) messages they exchange. Obviously, symmetric primitives make the assumption that the secret key is *securely* generated and exchanged among the sending and receiving parties prior to the encryption/decryption process. On the other hand, asymmetric cryptography (also called public key cryptography [60]) removes this requirement by *cleverly* generating two keys, such that if a message is encrypted by one key, it can only be decrypted by the other. Breaking public key schemes usually reduces to solving some hard mathematical problem.

Beside the symmetric and asymmetric primitives, there is another (controversial) class of cryptographic primitives that can be classified under both categories. Cryptographic hash functions (depicted in figure 1-1) are deterministic cryptographic primitives that transform an arbitrary size input into a fixed size output. Today, cryptographic hash functions have evidently become among the most active research areas in cryptography, spanning a wide space of applications from digital signatures [118] and key distribution schemes [102] to password protection [97]. Figure 1-2 illustrates the rapid growth that the literature of hash functions has witnessed in recent years[1] illustrating, in particular, the spike around 2005 when Wang *et al.* published their popular

[1]Figure 1-2 is based on estimated number of annual (not cumulative) hash functions publications.

attacks agains MD5 and SHA-1 [154, 155, 156], which later drove the U.S. National Institute of Standards and Technology (NIST) to announce their SHA-3 competition (more on this in chapter 3).

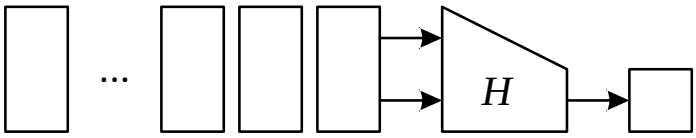


Figure 1-1: Sample hash function

On the one hand, it can be argued that hash functions are symmetric primitives because they are usually built from symmetric primitives such as block-ciphers and stream-ciphers, and in fact "any" block/stream cipher can trivially be used to build a hash function. Hash functions can also be keyed, in which case they clearly resemble symmetric primitives. On the other hand, hash functions are commonly used in conjunction with public key primitives, mostly with digital signatures. We will not delve into this argument, but we noticed that historically hash functions have been generally considered symmetric primitives.

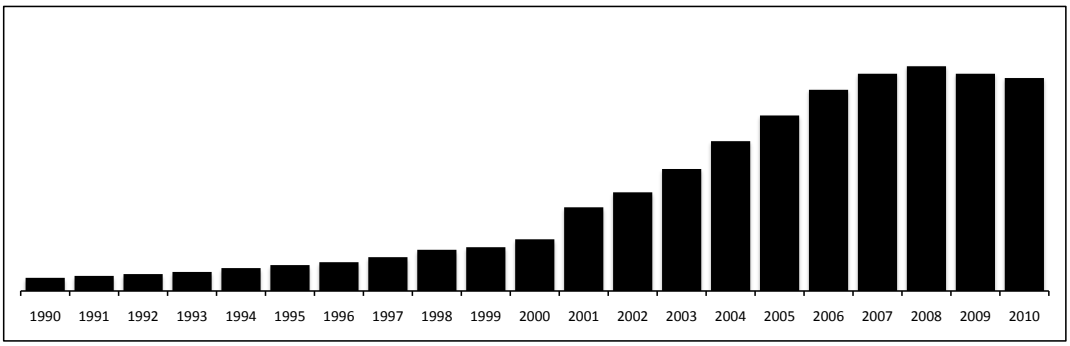


Figure 1-2: Growth of the cryptographic hash functions literature

The properties that typical cryptographic hash functions should preserve are usually application dependent, but it is widely accepted that cryptographic hash functions should be collision resistant, pre-image resistant and 2nd pre-image resistant.

- Collision resistance means that it should be infeasible for an adversary to find two different messages that would both hash to the same value.
- Pre-image resistance means that given a particular hash value, it should be infeasible for an adversary to invert it and find the message that produced it.
- 2nd pre-image resistance means that given a message and its hash value, it should be infeasible for an adversary to find a second message hashing to the same value.

However, collisions, pre-images and 2nd pre-images in hash functions are unavoidable due to the fact that these primitives are many-to-one functions. A "good" hash function would instead make finding collisions and (2nd) pre-images computationally infeasible[2].

For convenience, in the rest of the thesis we will often refer to cryptographic hash functions as simply hash functions, but they should not be confused with the non-cryptographic hash functions such as those used for table and database lookups, which are also called hash tables [95]. In such non-cryptographic applications, the hash function is exposed to much less stringent requirements than those that the cryptographic hash functions are expected to possess. For example, in non-cryptographic applications, it is sufficient that collisions in hash functions (two different messages producing the same hash value) are *rare*, but not necessarily computationally infeasible as the case with cryptographic hash functions.

1.1 Contributions

The main contribution of this thesis is the introduction, formalisation and security analyses of a (relatively) new class of cryptographic hash functions, named the integrated-key hash functions. Hash functions belonging to this class are said to have been constructed in the integrated-key setting. In this setting, keyless hash functions are seamlessly and transparently transformed into keyed hash functions and thus becoming families of hash functions. This conversion potentially strengthens the hash function and contributes towards more robust security arguments. Concisely, the contributions of the thesis can be summarised as follows:

- Introducing and formalising the integrated-key approach of designing hash functions, where keyless hash functions are transparently converted to keyed ones.
- Showing that the integrated-key setting is indistinguishable from its dedicated-key counterpart, where in the latter, keys are fed directly to the compression function. Thus, when converting a keyless hash function to a keyed variant in the dedicated-key setting, the internal structure of the compression function needs to be (sometimes significantly) modified to accommodate the key input. This result means that proofs of hash functions in the dedicated-key setting can be lifted and will still apply on their variants in the integrated-key setting.

[2] As a result of the rapid advances in computational resources, the term *computationally infeasible* is becoming somewhat vague. In fact, the amount of computation that might be considered infeasible today, may not be so in the near future. Currently, a computation with complexity 2^{128}, for example, can be safely considered infeasible, but this may cease to be the case at some point in the future according to Moore's Law [114], which states that the number of transistors that can be fitted in an integrated circuit doubles approximately every 2 years.

- Proposing three integrated-key hash constructions, after which the rest of the thesis is dedicated to analyse their security. While arguing about their security, we adopt generic proof approaches making the proofs applicable to other integrated-key hash functions with similar structures. In particular, we prove that our proposed integrated-key hash constructions are:
  - Collision resistant, pre-image resistance, and 2nd pre-image resistance (these are briefly described above, but more details are in section 2.2).
  - Indifferentiable from Random Oracle ($\mathcal{RO}$): it is infeasible for a distinguisher to be able to differentiate between our proposed constructions and a genuine $\mathcal{RO}$. In this case, the hash function is said to behave like a $\mathcal{RO}$.
  - Indistinguishable from a Pseudorandom Function (PRF): a distinguisher cannot distinguish between our integrated-key constructions and a random function (a randomly chosen function from the set of all functions having a particular domain and range), except with negligible probability.
  - Unforgeable when instantiated as MACs: if our integrated-key constructions are secretly keyed, it is infeasible for an adversary to generate a valid message-tag pair without access to the secret key.
- Integrated-key hash functions, like the dedicated-key hash functions, exhibit unavoidable efficiency loss due to the processing of extra key input. Therefore, as a marginal contribution, appendix A provides a general discussion about improving the efficiency of hash functions implementations and demonstrates how following some (even minor) optimisation techniques may significantly improve the efficiency of a hash function implementation. These optimisation guidelines are generic and applicable to any implementation, not necessarily hash functions or cryptographic implementations.

1.2 Notation

We already provided comprehensive lists of symbols and notation used throughout this thesis. Those lists were written for the convenience of the reader, below we elaborate.

General Notation. We use the notation $x \xleftarrow{\$} \mathcal{X}$ to indicate that a particular value is chosen uniformly at random from the set $\mathcal{X}$ and is assigned to the variable x. Similarly, by $x \xleftarrow{\$} \{0,1\}^n$ we denote the process of choosing the value of x uniformly at random from the set of all binary strings of size n. When we write $x \xleftarrow{\$} \mathcal{X}\backslash\{y\}$, we mean that the value of x is chosen uniformly at random from the set $\mathcal{X}$, excluding the value

y. By $y \xleftarrow{\$} A(x)$ we mean that the adversary A on input x outputs a value and assigns it to the variable y. If no input is specified, A generates the output without an input. A function is usually defined in terms of the spaces of its inputs and outputs, for example $H : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$ defines a function H with two inputs of lengths m-bit and n-bit, respectively, returning an output of length n-bit. We let $\{0,1\}^n \times \{0,1\}^n \equiv \{0,1\}^{m+n}$ and $\{0,1\}^m \times \{0,1\}^n \equiv \{0,1\}^n \times \{0,1\}^m$. We denote by $|M|$ the length of the string M and by $\langle M \rangle_\beta$ the β-bit encoding of the length of M. Concatenation of n blocks is denoted by $M_1 || \ldots || M_n$, but for convenience we may sometimes simply use $M_1 \ldots M_n$ or $M_1, \ldots, M_n$. If a message M consists of n blocks, we refer to the i-th block by $M[i]$ where $i \in \{1, 2, \ldots, n\}$. Slightly abusing the notation, we use M^i to refer to the sequence of blocks $M_1, M_2, \ldots M_i$ where $i \leq n$, this is not to be confused with exponentiation. Note that M^i is equivalently to $M[1 \ldots i]$. We denote by $M^{(i)}$ the input to the i-th application of a compression function (this notation is only used in chapter 6). We use $[X]^n$ and $[X]_n$ to refer to the most (respectively, least) significant n-bit of the string X, that is $[X]^n = x_i || x_{i+1} || \ldots || x_n$, where $n > i > 1$, and $[X]_n = x_1 || x_2 || \ldots || x_n$. The notation X^Y indicates that a component X has an oracle access to another component Y (oracle access here means that Y is a public component that replies to any query it receives). The conditional probability that a system F will output Y given the input X is denoted by $P^F_{Y|X}$. This is not to be confused with the conventional conditional probability notation $\Pr[Y|X]$ denoting the probability that event Y will occur given that event X has already occurred. The ceiling function is denote by $\lceil x \rceil$ where the value of x is rounded to the smallest integer larger than x. By $Dom(F)$ and $Rng(F)$ we denote the domain and range of function F, respectively. We denote a function taking a key K as an input by $g(K, .)$ or $g_K(.)$. The empty string is denoted by $\perp$. We refer to all values in the column $\hat{x}$ of the table $\mathcal{T}_K$ by $\mathcal{T}_K(\hat{x})$. We use the subscript _ (underscore) to index an arbitrarily chosen row in a table, e.g., $x_\_$ refers to the value of the x field of a random row in a table (table notation is only used in chapter 5). We use the symbols $\circ, \triangle$ to denote full and partial sequential composition while using $\star$ to denote an arbitrary composition operation. We also use the symbol $\bowtie$ to denote indistinguishability (see chapter 6). The notation $(\mathcal{X} \rightarrow \mathcal{Y})$-$\mathbf{R}$ indicates that the system $\mathbf{R}$ accepts input in the space $\mathcal{X}$ and returns output in the space $\mathcal{Y}$.

Games-playing Notation. An adversary A is a probabilistic algorithm that takes input, processes it, and either succeeds and returns an output or fails and terminates returning $\perp$. The advantage of A in the xxx game, denoted by $\mathbf{Adv}^{\text{xxx}}_A$ or $\mathbf{Adv}^{\text{xxx}}(A)$, is characterised by $(t_A, q_A, L, \epsilon_A)$-xxx, where t_A is the running time of A, q_A is the total number of queries A sends throughout the game, L is the length of the largest query

generated by A, and ϵ_A is the probability that A will succeed in the xxx game. Note that the advantage of a game is actually its success probability, hence $\mathbf{Adv}_A^{\text{xxx}} = \epsilon_A$. We will often be explicit with the advantage and denote it by $\mathbf{Adv}_H^{\text{xxx}}(A)$ which formalises the advantage of an adversary A in breaking the primitive H (usually a hash function in this thesis) in the xxx sense. Furthermore, the success probability (advantage) of A is taken over any probabilistic (random) choices that A makes during its execution, this is referred to as "the random coins of A" or "the random coins used by A".

1.3 Preliminaries

This section provides basic background of some mathematical concepts that will be used heavily throughout the thesis, but it is by no means comprehensive. The reader is advised to consult standard college textbooks for this purpose, e.g., [137].

Probability Theory. Probability theory is a fundamental security analysis tool in cryptography, used as a robustness measure of a system or algorithm against attacks. Considering the hash function $H : \mathcal{M} \rightarrow \mathcal{Y}$, we first give several definitions. The set $\mathcal{Y}$ is called the sample space, which is the set of all possible outcomes of an experiment. An experiment, in turn, is a procedure yielding a point in $\mathcal{Y}$ (in our scenario, the experiment is the actual hashing). An event is a subset of $\mathcal{Y}$ (i.e. one or more points from $\mathcal{Y}$). The complement of an event A is the set of all points in $\mathcal{Y}$ that do not belong to A, and is denoted by $\bar{A}$, that is $\bar{A} = \mathcal{Y} - A$. In cryptography, functions are usually defined on discrete spaces, that is, the sets $\mathcal{M}$ and $\mathcal{Y}$ above contain discrete points. Each outcome (point) in $\mathcal{Y}$ as returned by H occurs with a particular probability. *Random variables* are functions assigning probabilities to points of $\mathcal{Y}$ following a *probability distribution*. The most commonly used probability distribution in cryptography (and the one assumed throughout this thesis) is the uniform distribution which states that all outcomes of an experiment are equally likely, that is $P(y) = 1/|\mathcal{Y}|$. It is generally accepted that modern probability theory is based on three fundamental axioms:

1. the probability of events is non-negative: $P(y) \geq 0$ where $y \in \mathcal{Y}$,
2. the sum of the probabilities of all events is 1: $\sum_{i=0}^{|\mathcal{Y}|} P(y_i) = 1$, and
3. Additivity of disjoint events.

Two events, A and B, are said to be disjoint (or mutually exclusive or independent) if the occurrence of one does not influence the occurrence of the other, in which case the probability that both events occur at the same time is $P(A \wedge B) = P(A) \cdot P(B)$, while the probability that either (or both) events occur is $P(A \cup B) = P(A) + P(B)$.

Birthday Paradox. Consider the following surprising (and quite astonishing) fact:

> There is at least 50% chance that there will be two people having the same birthday in a group consisting of as little as 23 randomly chosen people, and a 99% chance for a group of only 57 people.

This paradoxical phenomena is called the Birthday Paradox. Formally, for a group of N people, the probability that there are at least two people having the same birthday is as follows (assuming that the birthdays of the N people are uniformly distributed over a "common year" whose days is 365):

$$P(N) = 1 - \prod_{i=0}^{N-1} \left(1 - \frac{i}{365}\right) = 1 - \frac{365!}{365^N (365 - N)!}$$

The birthday attack is based on the Birthday Paradox and entails choosing q random messages $x_1, x_2, \cdots, x_q$ from the domain D and compute their hashes $y_1, y_2, \cdots, y_q$, where $y_i = H(x_i)$ and H is a hash function. The birthday attack succeeds if there are $y_i = y_j$ while $x_i \neq x_j$, which is called a collision between x_i and x_j. That is, for a hash function $H : \mathcal{M} \to \{0,1\}^n$, where $\mathcal{M}$ is a message space and $|\mathcal{M}| > n$, a collision is expected to occur after q queries with probability:

$$P(q) = \binom{q}{2} \cdot \frac{1}{n^2} = \frac{q(q-1)}{2} \cdot \frac{1}{2^n} = \frac{q^2 - q}{2^{n+1}} \approx \frac{q^2}{2^n}$$

implying that a collision can be found after around $q \approx \sqrt{2^n} = 2^{n/2}$ messages chosen uniformly at random [29]. This attack can also be parallelised as discussed in [152], which significantly improves its efficiency.

However, as reported in [29], in order for this bound to hold, the hash function $H : \mathcal{M} \to \mathcal{Y}$ should be *regular*, which means that every point in the range $\mathcal{Y}$ should have the same number of pre-images in the domain $\mathcal{M}$ (recall that collisions are unavoidable as long as $\mathcal{M} > \mathcal{Y}$). Otherwise, collisions in H can be found faster than $q^2/2^n$. This, however, does not pose a significant practical concern, as it seems that most popular hash functions, e.g., MD5, SHA-1 etc., are "close enough" to being regular functions.

This attack, nonetheless, usually produces *meaningless* collisions because it searches for "any" two colliding messages. This behaviour may not have significant effect in practice as applications usually restrict the message scope. What and adversary would aim for is to produce *meaningful* collisions such that a collision finding algorithm returns two related messages, one seemingly "good" and another seemingly "bad", then the adversary can use these two messages interchangeably for malicious purposes. Examples from the literature demonstrated that such meaningful collisions are possible [145, 146].

1.4 Thesis Outline

This thesis consists of three main parts (excluding chapters 1, 7 and appendix A). Part I comprises chapters 2 and 3, and provides general background information on the state of the art of cryptographic hash functions. Part II comprises chapter 4, while part III comprises chapters 5 and 6, these two parts constitute the contributions of the thesis, where we introduce the integrated-key paradigm, propose several constructions in this setting and thoroughly analyse them.

- Chapter 1: Introduction. In this chapter we provide a brief and general (but informal) overview of cryptographic hash functions and their applications. We also discuss the contributions of the thesis and introduce the notation that will be used throughout the thesis. Finally, we describe how the rest of the thesis is organised, we give concise summaries of each chapter, and discuss how the chapters are related to each other.

- Chapter 2: Security of Hash Functions. In this chapter we take a close look at the various security notions and properties of hash functions. We provide both formal and informal definitions and discussions about the various security properties, such as collision resistance, pre-image resistance and 2nd pre-image resistance. We also provide extended discussions about several important notions such as the indifferentiability from Random Oracle ($\mathcal{RO}$), the indistinguishability from Pseudorandom Functions (PRF) and the unforgeability of Message Authentication Codes (MAC) algorithms. Most of these properties will be used in later chapters while analysing the security of our proposed integrated-key hash functions.

- Chapter 3: Design of Hash Functions. In this chapter, we provide a thorough discussion on the state of the art of hash functions design and what approaches were/are being adopted in constructing them. We describe the various design approaches and give examples from the literature. In particular, we discuss the popular Merkle-Damgård construction, describe the various generic attacks reported against it, and briefly discuss several variants of it showing how they were designed to thwart some of its generic attacks. Beside iterative hash functions, we also briefly discuss the parallel ones, though these are less common. Finally, we discuss how compression functions are being designed and describe several approaches.

- Chapter 4: Integrated-key Hash Functions. In this chapter, we introduce, define and formalise our integrated-key approach of constructing hash functions (where

we discuss how to seamlessly convert a keyless hash function to a keyed one without modifying the underlying keyless compression function). We propose three integrated-key hash constructions based on the Merkle-Damgård construction, and prove that they are collision resistant, pre-image resistant and 2nd pre-image resistant.

- Chapter 5: Indifferentiability of the iMD Constructions. Continuing our security analysis of the three integrated-key constructions proposed in chapter 4, in this chapter we further prove that these constructions are indifferentiable from $\mathcal{RO}$. We provide a detailed indifferentiability proof and show how such proofs are constructed in the integratd-key setting. Even though we explicitly developed the proof to argue about the indifferentiability of several specific constructions, the proof is generic and may be applicable to other hash functions constructed in this setting.

- Chapter 6: Indistinguishability and Unforgeability of the iMD Constructions. The contributions of the thesis are concluded in this chapter by first showing that hash functions constructed in the integrated-key setting are indistinguishable from their variants in the dedicated key setting, which (using the indifferentiability result in chapter 5) immediate implies that they are indistinguishable from Pseudorandom Functions (PRF). Using this indistinguishability result, we also prove that our proposed integrated-key constructions are unforgeable when used as MACs (i.e., secretly keyed). Unforgeability means that an adversary cannot output a valid message-tag pair, except with negligible probability.

- Chapter 7: Conclusion and Future Work. The thesis concludes in this chapter with final remarks and a few brief discussions about several possible interesting extensions to the work presented in this thesis.

- Appendix A: Engineering Aspects of Hash Functions. Finally, we provide an investigation in how to improve the efficiency of hash functions' implementations. We provide a set of optimisation guidelines to help implementers optimising their implementations. Although the discussion is mainly based on Intel platforms, most of the optimisation techniques are generic and platform independent.

At the beginning of each chapter, we provide a gentle overview of the chapter's contributions and what results from previous chapters are being used there; each chapter also concludes with a brief summary. Figure 1-3 depicts the dependency among the chapters. An arrow from chapter X to chapter Y means that results/definitions/notions from chapter X were used in chapter Y, then chapter X should preferably be read or

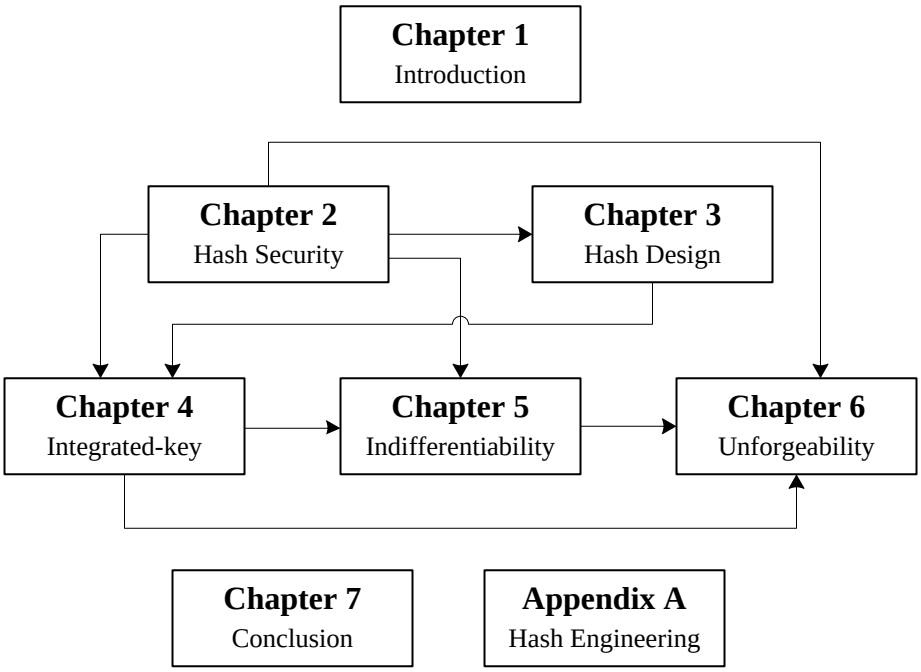


Figure 1-3: Dependencies among the chapters of the thesis

at least skimmed through before chapter Y. Apart from chapters 1, 7 and appendix A, most of the later chapters are built on definitions and results from earlier chapters. While chapters 2 and 3 present general background material, most of the details and notions discussed in these chapters are extensively utilised in the rest of the thesis as we develop the various security arguments and proofs. Chapter 4 provides definitions and results that would later be used in chapters 5 and 6, where chapter 6 also use results from chapter 5. Similarly, while chapter 5 does not directly use results from other chapters, it is based on our lengthy discussion of the indifferentiability framework in chapter 2 (specifically, section 2.3), so it is highly recommended to read that section before chapter 5. Parts of chapters 2 and 3 were published in [5] (extended version is in [6]). The material in appendix A was published in [2]. Papers forming the basis of chapters 4, 5 and 6 have been submitted for publication in peer-review venues [3], [4], and, at the time of writing, are still under review.

Chapter 2

Security of Hash Functions

The security of hash functions can be evaluated based on many criteria and is usually application dependent, where hash functions are expected to preserve a number of properties. However, if a particular hash function H failed to preserve a particular property X, that does not immediately render that hash function broken in practice, it just means that H cannot be "safely" used in applications where property X is required. In this chapter, we discuss the various hash functions security properties and notions. We also elaborate on how hash functions security proofs are developed in a provable security framework, which we would then use extensively in later chapters. The contents of this chapter was published in [5],[6].

2.1 Introduction

Hash functions are essentially many-to-one functions since they map arbitrary length inputs to fixed length outputs and the input is usually larger than the output (hash functions are compressing primitives). Thus, collisions (different messages hashing to the same value) in hash functions are unavoidable due to the pigeonhole principle[1]. Yuval [164] was the first to discuss how to find collisions in hash functions using the Birthday Paradox, leading to what is commonly known today as the *birthday attack*. In this attack, a collision is found with probability $q^2/2^n$ after q queries to a hash function outputting values of length n-bit [29]; see section 1.3 for more details about this attack. While collision resistance is certainly an important property that hash functions are expected to possess, it is not the only one, and in some applications it is not even required. Pre-image resistance (non-invertibility), for example, is a more

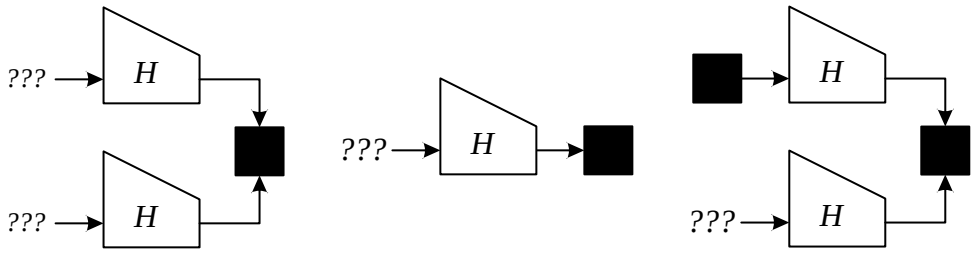
[1]The pigeonhole principle states that if m pigeons are distributed over n holes, and $m > n$, then there is at least one hole with more than one pigeon.

difficult and more practical property. In fact, in most applications it is more devastating to be able to invert a hash value, than finding a collision between two arbitrary messages. For example, applications such as password storage only require pre-image resistance since here the aim of an adversary is to reverse a given hash to obtain the corresponding password. Digital signatures is another example of applications that are only minimally affected by collision resistance, where an adversary not only required to generate meaningfully related colliding messages, but also sign them using the signer's private key, which should be accessible only to the singer. Therefore, the application that the hash function is targeted for determines the security properties that it should preserve. In this chapter, we present and discuss hash functions' most common security properties and notions. In preparation for the following chapters, we also discuss how hash functions security proofs are developed.

Chapter Outline. This chapter is organised as follows. First, in section 2.2 we provide a discussion about the three classical hash function security properties, namely collision resistance in section 2.2.1, pre-image resistance in section 2.2.2 and 2nd pre-image resistance in section 2.2.3, as well as some other properties (including statistical and application-specific ones) in section 2.2.4. We then describe in length the indifferentiability framework in section 2.3, which is later used in chapter 5. Brief descriptions of the indistinguishability from Pseudorandom Function (PRF) and the unforgeability notions are provided in sections 2.4 and 2.5, respectively. We also talk about some other security notions in 2.6. In section 2.7, we discuss the multi-property-preserving paradigm (MPP) and give a few examples of recent MPP constructions. Finally, we conclude this chapter in section 2.8 with a brief discussion on how proofs of hash functions are commonly developed in a provable security framework.

2.2 Classical Properties

The basic (classical) properties a hash function is expected to preserve are: collision resistance, pre-image resistance and 2nd pre-image resistance; figure 2-1 illustrates these properties graphically. Although these are thought to be the universal security properties that most hash functions should preserve, there may be other application-specific security properties that hash functions should additionally (or instead) preserve if they are to be used in a given application; here we provide an elaborate discussion on the basic definitions of these properties. In this chapter, and throughout the thesis, when we say that an attack succeeds in breaking a particular hash function, that does not necessarily mean that the hash function is deemed broken in practice. If an attack



(a) Collision resistance (b) Pre-image resistance (c) 2nd pre-image resistance

Figure 2-1: Graphical representation of the collision, pre-image and 2nd pre-image resistance properties

succeeds to prove that a hash function can be exploited (e.g., finding a collision, or pre-image, or second pre-image) with work less than that required by the birthday or brute force attack, the hash function is considered broken, even if the work required to break it is still infeasible in practice (this is sometimes called a theoretical break). Indeed, finding such flaws in a hash function is an evidence of structural weaknesses that may be exploited at later stages to turn this theoretical break into a practical one; the prime example is MD5, which was first theoretically broken, then the attacks eventually evolved and today practical collisions can easily be found in MD5 [94, 146].

2.2.1 Collision-Resistance (CR)

A hash collision occurs when two (arbitrary) different messages hash to the same value. That is, for a collision resistant hash function H, it should be computationally infeasible to find any two messages M and M' such that $H(M) = H(M')$ while $M \neq M'$. This also applies to families of hash function (i.e., keyed hash functions, where members of the family are indexed by different keys). Formally, the advantage of an adversary A of finding a collision in a hash function H is defined as follows:

$$\mathbf{Adv}^{\mathrm{cr}}_{H}(A) = \Pr\left[(M, M') \xleftarrow{\$} A : M \neq M' \wedge H(M) = H(M')\right]$$

For a secure hash function, the best collision finding attack should not be better than the birthday attack (i.e., not better than work complexity of $2^{n/2}$ for a hash function outputting n-bit hash values). Collision resistance was first formally defined by Damgård [54], and is sometimes called *Strong Collision Resistance*. Some authors use the term *multi-block collision* to refer to 2 colliding messages, each consisting of at least 2 blocks. This is not to be confused with multi-collision, where multiple messages collide at the same hash value regardless of their sizes (sometimes also called K-collision, where K is the number of the colliding messages). Finding a K-collision should have

a complexity of at least $2^{(K-1)n/K}$. In [134], Rogaway discussed the *foundation-of-hashing dilemma* which states that collision resistance cannot be formally defined for keyless hash functions, that is, there will always be collisions, it is just that us, humans, may not be able to find them, but such a concept cannot be formalised mathematically for keyless hash functions. Rogaway proposed a solution to this dilemma by means of explicit security reduction, which he called the *human-ignorance* framework.

2.2.2 Pre-image Resistance (Pre)

For all practical purposes, hash functions should be computationally *non-invertible*. When a message is hashed, it should be (computationally) infeasible to retrieve the original message from which the hash value was obtained. That is, for a pre-image resistant hash function H, given a hash value $H(M)$ of a particular message M, it should be computationally infeasible to retrieve the original message M, or indeed generate any message $M' \neq M$ such that $H(M') = H(M)$. Succinctly,

$$\mathbf{Adv}_H^{\mathrm{pre}[m]}(A) = \Pr\left[M \xleftarrow{\$} \{0,1\}^m; Y \leftarrow H(M); M' \xleftarrow{\$} A(Y) : H(M') = Y\right]$$

For a hash function to be pre-image resistant, the best attack against the hash function should be the brute force attack (i.e., work complexity of 2^n operations for a hash function with output size n). Pre-image resistance is also sometimes called *One-wayness*. Generally, collision resistance *does not* guarantee pre-image resistance [54], but in [135] it was shown that pre-image resistance can be implied by collision resistance if the hash function was sufficiently compressing (i.e., its domain is significantly larger than its range), otherwise if the hash function was length preserving, length increasing or compresses its input by only a few bits, the advantage of reversing its output significantly increases. Similarly, Stinson [147] argued that it is possible to obtain good reduction from collision resistance to pre-image resistance under several assumptions, but he then showed that these assumptions are difficult/impossible to satisfy in practice. For example, he showed that a collision resistant hash function is also pre-image resistant if the hash function is "close" enough to being uniform, but while uniformity of a function can be verified for random hash functions, it cannot be accurately verified for practical ones.

Stinson also introduced the zero pre-image notion, where an attacker finds a message M such that $H(M) = y = 0$. He remarked that there is no obvious reduction between zero-pre-image (where a specific value of y is inverted) and normal pre-image (where a random value of y is inverted), and that one may be harder or easier to find than the other.

2.2.3 2nd Pre-image Resistance (Sec)

Given a 2nd pre-image resistant hash function H, and a message M, it should be computationally infeasible for an adversary A to find a different message M' such that $M \neq M'$ and both M and M' hash to the same value, $H(M) = H(M')$. Succinctly,

$$\mathbf{Adv}_H^{\mathrm{sec}[m]}(A) = \Pr\left[M \xleftarrow{\$} \{0,1\}^m; M' \xleftarrow{\$} A(M) : M \neq M' \wedge H(M) = H(M')\right]$$

For H to be considered 2nd pre-image resistant, the best attack against H should be the brute force attack (i.e., work complexity of 2^n for a hash function with output size n). 2nd pre-image resistance is also sometimes called *Weak Collision Resistance*. Although it is frequently claimed in the literature that collision resistance implies 2nd pre-image resistance [110, 135], Contini *et al.* [47] argued that interpreting some formal definitions of 2nd pre-image resistance in the literature (e.g., [110]) invalidates this claim, but generally this claim is true if both collision and 2nd pre-image resistance are defined properly. Like collisions, a generalisation of 2nd pre-image is K-way 2nd pre-image where, given a message M and its hash value $H(M)$, an adversary A finds K different messages colliding at $H(M)$, that is, $H(M_1) = H(M_2) = \cdots = H(M_K) = H(M)$, while $M_1 \neq M_2 \neq \cdots \neq M_K \neq M$. Similarly, K-way pre-image occurs when, given $H(M)$, an adversary A finds K different messages colliding at $H(M)$. Finding a K-way (2nd) pre-image should have a complexity of at least $K.2^n$.

Remark. In [160], Yasuda showed that the compression function of a Merkle-Damgård hash function (see section 3.3.1) does not have to be CR for the whole hash function to be Sec or Pre. The author argued that it only suffices for a compression function to preserve weaker-than-CR properties, namely cs-SPR (chosen suffix second pre-image) and cs-OW (chosen suffix one-wayness), for the corresponding hash function to preserve Sec and Pre, respectively. This is indeed an important result since many compression functions used with the Merkle-Damgård construction were recently found to be not CR, which directly implies that their corresponding hash functions are not CR [111, 55], but that does not necessarily mean that they are also not Sec and/or Pre when their compression functions are modelled as cs-SPR and cs-OW (which are weaker than CR).

2.2.4 Other Properties

Other desirable properties that hash functions should preferably preserve include [110] (some of these properties are application specific, i.e., some hash function applications may not require some of these properties):

- Near-collision resistance: hash values of two different messages should differ significantly (even if the messages that produced them differ slightly), that is, a near-collision occurs if for two different messages $M \neq M'$, then $H(M)$ differs from $H(M')$ by only a small number of bits.

- Semi-free-start collision resistance: a semi-free-start collision occurs when two different messages with the same (but random) IV hash to the same value. In practice, though, hash functions are usually specified with fixed IVs.

- Pseudo-collision resistance: a pseudo-collision (or free-start collision) occurs when it is possible to find a collision between two messages by only controlling their IVs (again, this attack is not practically relevant because most hash functions are shipped with fixed IVs, so an attacker cannot control the IV in practice). A variant of this property is pseudo-near-collision, which results in a near-collision.

- Partial pre-image resistance: also sometimes called local one-wayness, states that it should be equally difficult to retrieve part of the original message from its hash value as retrieving the whole message, even if a portion of the message is already known (in chapter 4, we will introduce full and partial inversion, which are similar to pre-image and partial pre-image).

- Non-correlation (correlation freeness): hash function inputs and outputs should not be statistically correlated; that is, even a small change in the input should drastically affect the output bits; this phenomenon is called the avalanche effect.

- Chosen Target Forced Prefix (CTFP) pre-image resistant [91]: applications that need to resist the herding attack [91] (see section 3.3.2) need to preserve this property, which prevents an attacker from finding a string S such that given P and H, then $H(P||S) = F$ where F is a hash value computed before learning P.

A particularly problematic situation arises when trying to evaluate the security of sponge-based constructions [33] (see section 3.3.4). Traditionally, security bounds are based on the function's output length n, where collision requires $2^{n/2}$ and pre-image/2nd pre-image require 2^n. However, the sponge construction produces a variable length output. Realising this problem, the authors of the sponge construction introduced a reference security model, called the *Random Sponge*, which they use in their security analysis. Detailed discussion of the security of the sponge construction is beyond the scope of this thesis.

2.3 Indifferentiability from $\mathcal{RO}$

Security analysis and proofs of many cryptosystems are carried out in the Random Oracle Model (ROM) [26], which assumes the presence of a *Random Oracle* ($\mathcal{RO}$). A $\mathcal{RO}$ is an abstract ideal primitive that returns infinite random response every time it is queried [26], though such response is usually truncated. Responses of $\mathcal{RO}$ are consistent for similar queries (a particular query will always receive the same $\mathcal{RO}$ response regardless of when and how many times it is made) and since $\mathcal{RO}$ is an atomic entity (i.e., it cannot be decomposed), it is often said to be *monolithic*. In practice, $\mathcal{RO}$s do not exist [40] and are instead instantiated by hash functions which are *not* monolithic by nature since hash functions are structured entities that usually process messages by repeatedly and iteratively calling an underlying primitive (commonly, a compression function). In particular, in [40] Canetti *et al.* showed that there exist schemes that are secure in the ROM but become insecure when the $\mathcal{RO}$ is replaced by "any" practical implementation. Therefore, for proofs in the ROM[2] to hold in practice, the adopted hash function should *emulate* a $\mathcal{RO}$. A hash function H emulating a $\mathcal{RO}$ in this sense implies that H cannot be "differentiated" from a genuine $\mathcal{RO}$ and that systems proven secure in the ROM will remain secure if the $\mathcal{RO}$ is replaced by H.

Based on Maurer's indifferentiability framework [108], Coron *et al.*, introduced their popular hash function indifferentiability from $\mathcal{RO}$ framework in [48], which can be used to prove that a hash function, with access to an ideal compression function, is indifferentiable from a genuine $\mathcal{RO}$. In this framework, the two building blocks of a hash function, namely a construction C and an ideal compression function $\mathcal{G}$, constitute a system (System 1) and the random oracle $\mathcal{F}$ (which the hash function needs to emulate) constitutes another system (System 2), then a distinguisher algorithm D with oracle access[3] to both systems tries to challenge the systems and distinguish between them[4]. If we do not introduce an extra component in System 2, D can easily distinguish between the two systems since System 1 consists of two components while System 2 consists of only one component. Thus, we introduce a simulator S in System 2 to simulate the ideal

[2]Even though proofs in the ROM may not *always* guarantee security when the $\mathcal{RO}$ is instantiated by a practical hash function, if a scheme is proven secure in the ROM, there is strong "heuristic" evidence that this scheme exhibits sound structure, or at least do not suffer from serious inherent structural weaknesses.

[3]When a X has oracle access to Y, X can publicly query Y and receive response back. However, oracle access does not necessarily imply a black-box access, where in the latter X can only query Y, but cannot access its internal components (if there is any).

[4]In the context of cryptography, there is a distinction between *indifferentiability* and *indistinguishability*. In indistinguishability, a distinguisher algorithm D is merely given black-box access to the two systems. In indifferentiability, on the other hand, D is further given access the underlying primitives of the systems and can query them independently. Thus, indifferentiability is clearly a generalisation of indistinguishability.

compression function $\mathcal{G}$ of System 1. The simulator S should be defined very carefully to simulate not only a conventional compression function operation (such that given an input it returns a compressed random output), but also how to behave consistently with the way C and $\mathcal{G}$ interact to handle D's queries. This is indeed a non-trivial task for S because all messages sent to C will be processed by $\mathcal{G}$ (i.e., $\mathcal{G}$ can see all messages sent to System 1, including those sent to C), but that does not hold for S in System 2 because the random oracle $\mathcal{F}$ is an atomic component and will return its responses independently from S (i.e., messages sent to $\mathcal{F}$ are not observable by S). It is important to note that here D challenges the systems by sending multiple queries to the different components of the systems and observes the responses, D then distinguishes between the systems based on their overall observed behaviour not just the individual responses of the queries. That is, if D sent a query to both systems and received different responses, that does not necessarily mean that D has succeeded in the distinguishing game because both responses are still random. However, if D observed a pattern in a series of responses from a particular system but did not observe similar behaviour in the other system, then it is apparent that one of them is behaving differently than the other and D succeeds in the distinguishing game. Although D does not necessarily have to tell which system is which, it will be obvious that the system that behaves in a more random manner is the $\mathcal{RO}$ system. Figure 2-2 illustrates the general setting where D has oracle access to both Systems 1 and 2.

Systems 1 and 2 are also sometimes called the *Real System* and the *Random System*, respectively. In System 1, C has oracle access to $\mathcal{G}$, while in System 2, S has oracle access to $\mathcal{F}$. However, whereas in System 1, C always queries $\mathcal{G}$ to obtain responses for any query it receives, in System 2, S may choose to query $\mathcal{F}$ or generate its responses uniformly at random. For Systems 1 and 2 to be indifferentiable from each other, it is important that S is programmed in such a way that $\mathcal{F}$ and S behave consistently with how C and $\mathcal{G}$ behave (note that S is the only customisable component). To illustrate this point, let's look at how $\mathcal{G}$ (which S should simulate) behaves. When $\mathcal{G}$ receives a query (which may be from C or D) it merely generates a random response since it is modelled as an ideal primitive. Having to simulate $\mathcal{G}$, the simulator S, in turn, should do the same, but S is not an ideal primitive, so it can either query $\mathcal{F}$ to get a random oracle response (i.e., a response from an ideal primitive), or just return a uniformly random response from some randomness generation source. Since it is usually more expensive to query $\mathcal{F}$, S will always return uniformly random responses, unless it really has to query $\mathcal{F}$; when exactly to query or not to query $\mathcal{F}$ is (the main) part of the simulator's definition and depends on how System 1 behaves. In Merkle-Damgård-like hash functions (when C is modelled as a Merkle-Damgård construction), the main

differences between System 1 and System 2 can be summarised as follows:

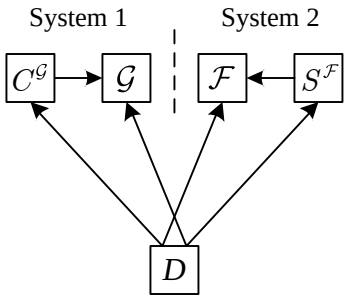


Figure 2-2: Distinguisher's view in the indifferentiability game

1. C vs. $\mathcal{F}$: since C is an iterative construction, it processes its input as a sequence of blocks which are sent to $\mathcal{G}$ in turn; however, this is not the case with $\mathcal{F}$ because $\mathcal{F}$ processes the whole message (no matter how long it is) independently at once.

2. $\mathcal{G}$ vs. S: it is clear that $\mathcal{G}$ will necessarily be aware of all queries sent to System 1 because C will eventually process its queries through $\mathcal{G}$, but S, on the other hand, cannot see queries sent to $\mathcal{F}$, and thus may not be aware of them.

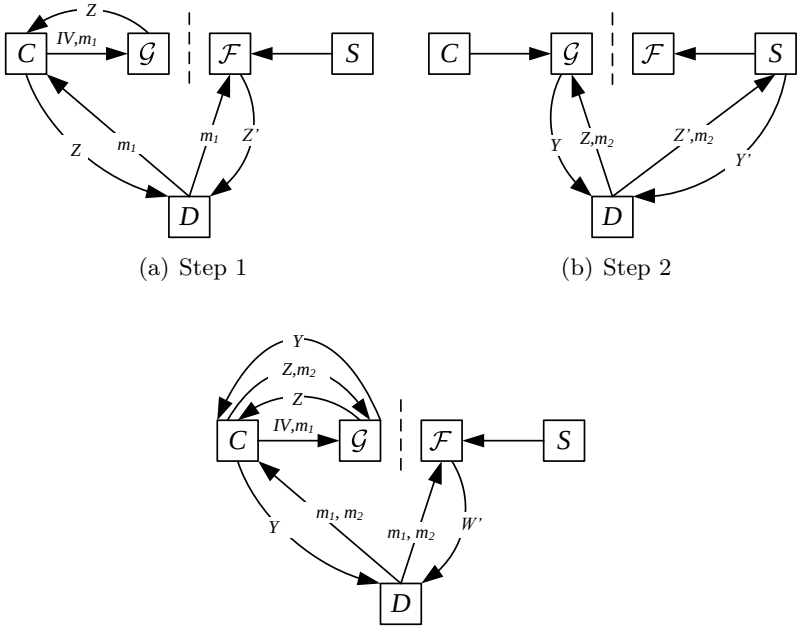
It is, therefore, the job of the simulator S to account for these differences to resist D's distinguishing attacks. However, this is only possible if C is a good construction; in fact, there are cases where D will always succeed in exploiting these structural differences no matter how intelligent and efficient S is. To illustrate how D can typically exploit the differences between the systems and distinguishes between them, we describe an attack against the popular Merkle-Damgård construction [111, 55] as reported in [48], which shows that the plain Merkle-Damgård construction is not indifferentiable from $\mathcal{RO}$; figure 2-3 illustrates the steps of the attack (see section 3.3.1 for a description of the Merkle-Damgård construction). Basically, here D exploits the fact that C processes its queries iteratively by calling $\mathcal{G}$, and that S cannot see queries sent to $\mathcal{F}$ (the differences listed above). The attack proceeds in three steps:

1. First, D sends the query m_1 to both C and $\mathcal{F}$ and receives $C(\mathcal{G}(IV, m_1)) = Z$ and $\mathcal{F}(m_1) = Z'$; the IV is hardcoded at C and added automatically.

2. Then, D sends the 2-block queries $Z||m_2$ and $Z'||m_2$ to $\mathcal{G}$ and S, respectively, and receives $\mathcal{G}(Z, m_2) = Y$, and $S(Z', m_2) = Y'$; here it will not make a difference whether S used F to generate Y' or it generated it uniformly at random.

3. Finally, D sends the 2-block query $m_1||m_2$ to C and $\mathcal{F}$ and receives $C(m_1, m_2) = C(\mathcal{G}(\mathcal{G}(IV, m_1), m_2)) = C(\mathcal{G}(Z, m_2)) = Y$ and $\mathcal{F}(m_1, m_2) = W$.

D then outputs 1 (i.e., success) in System 1 if $\mathcal{G}(Z, m_2) = C(m_1, m_2)$, and 0 otherwise (note that D decides on the success conditions which may be different when D programmed at different settings). Similarly, D outputs 1 in System 2 if $S(Z', m_2) = \mathcal{F}(m_1, m_2)$. It is easy to see that D will always output 1 when interacting with System 1, but will output 0 with overwhelming probability when interacting with System 2 (this is because S cannot see m_1, so it can only guess it, but this has a low probability of success). In this case, D succeeds in its distinguishing game. In summary, proofs in



(a) Step 1

(b) Step 2

(c) Step 3

Figure 2-3: Indifferentiability attack against the Merkle-Damgård construction

the indifferentiability framework proceeds in two steps:

1. First, we define a simulator S in System 2 (the Random System) to play the role of $\mathcal{G}$ in System 1 (the Real System), and whose relationship to the random oracle $\mathcal{F}$ mimics that of $\mathcal{G}$ to C.

2. Then, we prove that the view of a distinguisher D is similar when it interacts with the system consisting of the random oracle $\mathcal{F}$ and the simulator S (System 2), and the system consisting of the construction C and the ideal primitive $\mathcal{G}$ (System 1), except with negligible probability. Formally, the success advantage of an adversary D in the indifferentiability game (also called the Pseudorandom

Oracle (PRO) game [24]) is as follows:

$$\mathbf{Adv}_{C}^{\text{pro}}(D) = \left|\Pr\left[D^{C^{\mathcal{G}},\mathcal{G}} \rightarrow 1\right] - \Pr\left[D^{\mathcal{F},S^{\mathcal{F}}} \rightarrow 1\right]\right|$$

This setting can also be generalised for a construction accessing more than one primitive, in which case multiple simulators will need to be introduced in System 2 (Random System); see chapter 4.

The $\mathcal{RO}$ is sometimes realised as a pseudorandom oracle (PRO) [24], which is computationally indistinguishable from a genuine $\mathcal{RO}$ (we will use $\mathcal{RO}$ and PRO interchangeably to mean the same primitive). When a construction is proven to be indifferentiable from a $\mathcal{RO}$, it is sometimes referred as PRO-Pr (Pseudorandom Oracle Preserving).

2.3.1 Game-playing

The game-playing technique was first used in [93], then formalised in [28], and has been a popular technique for analysing cryptographic primitives ever since. This technique is usually used when we need to prove that two systems cannot be distinguished from each other. Let these systems be System 1 and System 2. We start with System 1 and write it as a *game* (pseudocode), we then introduce minor syntactical modifications to the game and calculate the probability that a distinguisher will be able to distinguish between the original game and the one with the minor modifications. The probability is usually calculated by introducing a flag `bad` that is originally set to false, and upper bounded by the probability that a distinguisher will set `bad` = true, where it succeeds in its distinguishing attack. We then keep introducing similar minor syntactical changes to the games and track the probability that the distinguisher will set `bad` = true between the consecutive games. The simulation ends when we reach the game that is identical to System 2. The overall distinguishing advantage is then easily upper bounded by recalling all the probabilities of setting `bad` = true. This technique is primarily used in the indifferentiability and indistinguishability proofs, e.g., [48], and indeed is being extensively used in chapter 5 (though, we make the use of a the flag `bad` implicit).

2.3.2 Salvaging differentiable Constructions

A serious problem arises when trying to use Coron's indifferentiability framework with hash functions based on mathematical primitives (such as those presented in section 3.5.2) because they are easily differentiable from $\mathcal{RO}$ due to the rigorous mathematical structure they exhibit (whereas $\mathcal{RO}$ are unstructured entities). However, while it seems that this class of hash functions *cannot* be considered practical hash functions (since

they are clearly differentiable from $\mathcal{RO}$), Ristenpart and Shrimpton [131] pointed out (and proved) that such hash functions can be slightly modified to be indifferentiable from $\mathcal{RO}$. They proposed a construction called Mix-Compress-Mix (MCM), which basically wraps the hash function with two injective mixing steps to mix the input and the output of the hash function and *hide* its (mathematical) structure.

In [121], the authors adopted a slightly different approach which gives hope to the constructions that failed to be indifferentiable from $\mathcal{RO}$. Their approach involves introducing weaker $\mathcal{RO}$ variants and then proving that the constructions that failed to be indifferentiable from $\mathcal{RO}$ are indifferentiable from these weaker $\mathcal{RO}$ variants. Cryptosystems, such as FDH, OAEP and RSA-KEM, are then secure under those constructions if they are secure in these weaker $\mathcal{RO}$ variants. This approach was demonstrated on the Merkle-Damgård construction which was shown to be not indifferentiable from $\mathcal{RO}$ in [48]. The authors proposed three $\mathcal{RO}$ variants as follows (in descending order from strongest to weakest): Leaky $\mathcal{RO}$ (LRO)[5], Traceable $\mathcal{RO}$ (TRO) and Extension attack simulatable $\mathcal{RO}$ (ERO). The authors proved that FDH is secure in LRO, OAEP is secure in TRO, and RSA-KEM is secure in ERO, then they proved that Merkle-Damgård is indifferentiable from LRO, TRO and ERO, which means that Merkle-Damgård is secure in FDH, OAEP and RSA-KEM.

2.4 Indistinguishability from PRF

In the keyed setting, a hash function is indistinguishable from Pseudorandom Function (PRF) if there is no adversary able to distinguish it from a random function [76]. A random function is a function that has been chosen randomly based on a given domain and range; this does not imply that the output of the function should be random, the randomness here refers to the function selection process not the output of that function. Indeed, the function with constant output (e.g., always outputs 1) is a random function if it was selected randomly. Being a random function (or indistinguishable from one) is an idealisation of hash functions because when a hash function is modelled as a random function $H_K : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{Y}$, every key $K \in \mathcal{K}$ should trigger the selection of a random function (member of the function family) that maps an input $M \in \mathcal{M}$ from the domain to a random output in the range $Y \in \mathcal{Y}$. Given such hash function family, it should be infeasible for an adversary A with black-box access to H_K to distinguish a randomly

[5]The LRO was proposed earlier by Yoneyama [163], also called pub-RO, in [64].

chosen member of H_K from a genuinely (pseudo) random function. Succinctly,

$$\mathbf{Adv}^{\text{prf}}_{H_K}(\mathrm{A}) = \left|\Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathrm{A}^{H_K} \rightarrow 1\right] - \Pr\left[R \xleftarrow{\$} Func(Dom, Rng) : \mathrm{A}^{R} \rightarrow 1\right]\right|$$

where $Func(Dom, Rng)$ denotes the set of all functions mapping inputs from the domain Dom to outputs in the range Rng, with R a randomly chosen function from such a set. In this setting, adversary A has access to both a family of hash functions H_K and a genuine (pseudo) random function R, both mapping inputs from the domain Dom to the range Rng. Adversary A queries one of the components (which may equally be H_K or R) and outputs 1 if it thinks that it is interacting with H_K, for example, 0 otherwise. A then repeats this process when it interacts with the other component and succeeds if it can make many correct guesses. Like the PRO game, in the PRF game the adversary A's view should be similar when it interacts with a PRF (or a random function) as it is when it interacts with the hash function H, where the latter selects its keys (members) uniformly at random; that is, A cannot distinguish between H and a PRF, except with negligible probability. A hash function proven to be indistinguishable from PRF is sometimes referred to as PRF-Pr (Pseudorandom Function Preserving). Note that Indistinguishability from PRF and unforgeability (section 2.5) only make sense in the keyed setting.

2.5 Unforgeability

MACs (Message Authentication Code) are popular cryptographic primitives used for authentication and integrity checks. One of the most established approaches of designing MACs is based on keyed hash functions[6], where the hash function is secretly keyed (and only legitimate parties possess the key). That is, both the sender and the receiver share a secret key (that is assumed to have been exchanged securely) which they use in conjunction with a hash function to check the integrity and authenticity of a particular message. Precisely, a MAC scheme consists of a tag generation algorithm and a tag verification algorithm. The sender uses the tag generation algorithm to generate a tag for a particular message (using its secret key), then it sends the message and the tag to the receiver. Once the receiver receives the message-tag pair (which may have been tampered with en-route), it runs them through the tag verification algorithm using its own secret key. If the message-tag pair is valid, the tag verification algorithm returns 1 and the message is authenticated, otherwise it returns 0 and the message is rejected/discarded. Note that here the tag verification algorithm basically just runs

[6]Another very popular approach is designing MAC based on block-ciphers [65], but block-cipher based MACs are beyond the scope of this thesis.

the tag generation algorithm on the message and the receiver's secret key, then compares the tag it generates with the tag the receiver received. Essentially, MACs are secret-key primitives, the counterpart of MACs in the public-key setting is digital signatures, where the sender uses its own private key to sign a message, which can then be verified using its public key. Digital signatures generally provide better authenticity than MAC because in MAC any party that is able to verify a tagged message can also produce a similar one, which means that if the secret key used to tag a message is shared by multiple parties, a particular tagged message could have been produced by any one of them. On the other hand, in digital signatures, once a message is signed by a sender, it is bound to that particular sender and no one else could have signed it because (presumably) no one else has access to the sender's secret key. Thus, digital signatures preserve an important property called *non-repudiation*, where a signer of a message cannot deny signing it. Digital signatures is yet another central applications of hash functions, but we do not address digital signatures in this thesis. For a hash function to be a good MAC, it needs to be unforgeable. Given an adversary (forger) A, the formal unforgeability definition is as follows:

$$\mathbf{Adv}^{\mathrm{mac}}_{H_K}(A) = \Pr\left[K \xleftarrow{\$} \mathcal{K}, (M,T) \xleftarrow{\$} A^{H_K} : H_K(M) = T \wedge M \text{ is not queried}\right]$$

The advantage definition of MAC implies that it should be difficult for A (a forger) to find a valid pair of a message M and a tag T which can then be successfully validated by a MAC algorithm (this message-tag pair is called a MAC forgery). A's aim is to find *any* valid pair of M and T (it is not about recovering the secret key K that the MAC algorithm used while generating T). That is, A builds the forgery pair (M', T') by repeatedly querying H_K with a set of adaptively chosen messages $M_1, \ldots, M_s$ and observes the returned tags $T_1, \ldots, T_s$, then A succeeds if it can generate a new message $M' \notin \{M_1, \ldots, M_s\}$ and a valid tag T' such that $H_K(M') = T'$.

Another, slightly non-standard, approach is to prove that a MAC algorithm is indistinguishable from an ideal MAC function (an ideal primitive), where the latter basically behaves as a random mapping from the domain (the function's input, i.e., the message and the key) to the range (the function's output, i.e., the tag) [71]. In this thesis, we will consider the pervious, more standard approach.

2.6 Other Notions

It has also been suggested that hash functions should behave as a randomness extractor (extracting uniformly random bits from an input generated by imperfect randomness

source) [61, 62]. However, in order for a hash function to possess randomness extraction properties, it may be necessary to make strong assumptions on the compression function that may not even be practically relevant.

Another notion is PrA (Pre-image Awareness) proposed by Dodis *et al.* in [62], which states that if an attacker can find a pre-image M of a previously published hash value Y, then it must have already known (was aware of) M; that is, a hash function is PrA if there is no adversary that can find a pre-image of a previously published hash value, unless it is already aware of that pre-image (Dodis *et al.* showed that strengthened Merkle-Damgård preserves PrA). A strengthened variants of PrA, called adaptive pre-image resistance [98], allows the adversary to make adaptive queries to the underlying compression function. It was shown that a collision resistant compression function that preserves the adaptive pre-image resistance property can yield a hash function indifferentiable from $\mathcal{RO}$.

2.7 Multi-Property-Preserving

One would naturally think that having a hash function provably preserving some strong security property (such as indifferentiability from $\mathcal{RO}$) is enough to imply a sufficient security margin. However, in [24] Bellare and Ristenpart refuted this assumption by providing counterexamples showing that the constructions proposed by Coron *et al.* in [48] are in fact *not* collision resistant while they are still indifferentiable from $\mathcal{RO}$, a supposedly strong security notion[7]. Thus, the authors suggested that a reasonably secure hash function should be *multi-property-preserving* (MPP).

In [135], Rogaway and Shrimpton provided a formal discussion about the relations between collision resistance, pre-image resistance, 2nd pre-image resistance and several variants of the latter two. They considered families of hash functions (keyed hash functions) while studying these properties, because the keyed setting is easier to formally analyse than the keyless setting [134]. These properties are CR, Pre, Sec, aPre, ePre, eSec, aSec, whose advantages are as follows (keyless CR, Pre, Sec were extensively discussed in section 2.2, here we repeat their advantages in the keyed setting):

[7] However, that does not degrade the importance of the indifferentiability as a property.

$$\mathbf{Adv}^{\mathrm{cr}}_{H_K}(\mathrm{A}) = \Pr\left[K \xleftarrow{\$} \mathcal{K}; (M, M') \xleftarrow{\$} A(K) : M \neq M' \wedge H_K(M) = H_K(M') \right]$$

$$\mathbf{Adv}^{\mathrm{Pre}[m]}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} K \xleftarrow{\$} \mathcal{K}; M \xleftarrow{\$} \{0,1\}^m; \\ Y \leftarrow H_K(M); M' \xleftarrow{\$} A(K, Y) : H_K(M') = Y \end{array} \right]$$

$$\mathbf{Adv}^{\mathrm{aPre}[m]}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} (K, St) \xleftarrow{\$} A(); M \xleftarrow{\$} \{0,1\}^m; \\ Y \leftarrow H_K(M); M' \xleftarrow{\$} A(Y, St) : H_K(M') = Y \end{array} \right]$$

$$\mathbf{Adv}^{\mathrm{ePre}}_{H_K}(\mathrm{A}) = \Pr\left[(Y, St) \xleftarrow{\$} A(); K \xleftarrow{\$} \mathcal{K}; M' \xleftarrow{\$} A(K, St) : H_K(M') = Y \right]$$

$$\mathbf{Adv}^{\mathrm{Sec}[m]}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} K \xleftarrow{\$} \mathcal{K}; M \xleftarrow{\$} \{0,1\}^m; \\ M' \xleftarrow{\$} A(K, M) : M \neq M' \wedge H_K(M) = H_K(M') \end{array} \right]$$

$$\mathbf{Adv}^{\mathrm{aSec}[m]}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} (K, St) \xleftarrow{\$} A(); M \xleftarrow{\$} \{0,1\}^m; \\ M' \xleftarrow{\$} A(M, St) : M \neq M' \wedge H_K(M) = H_K(M') \end{array} \right]$$

$$\mathbf{Adv}^{\mathrm{eSec}[m]}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} (M, St) \leftarrow A(); K \xleftarrow{\$} \mathcal{K}; \\ M' \xleftarrow{\$} A(K, St) : M \neq M' \wedge H_K(M) = H_K(M') \end{array} \right]$$

Let xxx $\in$ {Pre, Sec}, then saying that H_K is a-xxx means the hash function H_K is *always* xxx-resistant for a fixed key K and random challenge, while e-xxx means the hash function H_K is *everywhere* xxx-resistant for a fixed challenge and random key K. The challenge in Sec is the original message M (domain point) to which we need to find a 2nd pre-image, while it is the hash value $H_K(M)$ (range point) in Pre which we need to invert to find M. Occasionally, the adversary A may return a state variable St, which contains information that the adversary may need in later stages of the attack (this is how extra information is usually modelled in formal definitions). For example, if A generates a key K, A may wish to keep track of any random choices he made during the generation of K, so A stores this information in St.

The eSec property is also called Target Collision Resistance (TCR) [27] which is, in turn, another name for the popular Universal One Way Hash Function (UOWHF)[8] notion of Naor and Young [116]. Strengthened variants of some of these properties

[8] In UOWHF (or TCR or eSec), an adversary A generates a message M, then given a random key K, A generates another message $M' \neq M$, such that $H_K(M) = H_K(M')$. This is not to be confused with universal hash functions where A chooses both M and M' before it knows K. A strengthened variant of TCR is eTCR where A aims to find $M \neq M'$ such that $H_K(M) = H_{K'}(M')$ and $K \neq K'$.

were proposed in [78, 160, 130], namely s-CR, s-Sec, s-aSec, s-eSec, s-Pre, s-aPre (it was argued that ePre cannot be strengthened because if it was strengthened, then there will always be a trivial adversary succeeding in the s-ePre game).

Backward Chaining Mode (BCM) proposed by Andreeva and Preneel in [15] preserves the three classical security properties of hash functions, namely CR, Pre and Sec. Similarly, in [14] Andreeva *et al.* proposed the ROX (Random Oracle XOR) construction which preserves seven properties: CR, Pre, ePre, aPre, Sec, eSec and aSec. However, later work by Reyhanitabar *et al.* [129] showed that ROX is surprisingly not indifferentiable from $\mathcal{RO}$ even though ROX uses two $\mathcal{RO}$s in its padding algorithm. Moreover, in [25] Bellare and Ristenpart proposed the ESh (Enveloped Shoup) construction that preserves: CR, eSec, PRO-Pr, PRF-Pr and MAC, but later Reyhanitabar *et al.* [129] showed that ESh does not preserve Sec, aSec, Pre and aPre.

2.8 Cryptographic Proofs

There are two popular general approaches that proofs of cryptographic hash functions usually adopt, either constructing the proof in the standard model, or in the ideal model (regardless of whether the hash function was keyless or keyed). Proofs in the standard model assume the presence of primitives preserving/possessing standard (practical) properties, such as collision resistance, pre-image resistance, 2nd pre-image resistance etc. A standard model proof is then developed to argue that a hash function preserves such properties if it is given access to (or built from) primitives preserving those properties. For example, the hash function H is said to be xxx-secure if we can construct a standard model proof that demonstrates the following:

> *If a hash function H^G has oracle access to a standard primitive G, and G is* xxx*-secure, then H^G is also* xxx*-secure.*

Such proofs are said to be constructed in the standard model. On the other hand, proofs in the ideal model assume the presence of ideal primitives and proceed by proving that if a hash function has oracle access to (or built from) such primitives, it possesses a particular property (or set of properties). For example, a hash function H is said to possess the property xxx if a proof can be develop to argue about the following:

> *If a hash function $H^{\mathcal{G}}$ has oracle access to an ideal primitive $\mathcal{G}$, then $H^{\mathcal{G}}$ provably possesses the property* xxx*, or indifferentiable from $\mathcal{G}$.*

The ideal primitive $\mathcal{G}$ can be any idealised component, such as an ideal permutation, but the most commonly used ideal primitive in cryptography (in generally) is a Random

Oracle ($\mathcal{RO}$), which we discussed fairly thoroughly in section 2.3. Thus, the Random Oracle Model (ROM) can be thought of as the most popular ideal model, but it is not the only one. In fact, in chapter 5, we will develop our indifferentiability proof by adopting the Ideal Cipher Model (ICM), which assumes the presence of an ideal block-cipher since hash function are usually (directly or indirectly) built from block-ciphers.

While proofs in the standard model are more practically relevant, it is sometimes extremely difficult to carry out such proofs for some schemes, potentially making the ideal model the only (feasible) option. Clearly, nonetheless, proofs in the ideal model provide weaker security guarantees because they assume the presence of ideal primitives that may not exist in practice. In both approaches (and in this thesis), the adversary is usually assumed to have finite computational resources.

2.9 Summary

In this chapter, we discussed (both formally and informally) the most popular hash functions security properties and notions. We elaborated on the three classical notions of collision resistance, pre-image resistance and 2nd pre-image resistance, then discussed other properties such as near-collision resistance and pseudo-collision resistance. We then provided a lengthy discussion on the indifferentiability from Random oracle ($\mathcal{RO}$) framework and showed how proofs in this framework are generally structured. In the keyed setting (where hash functions accepts a key beside the message), a hash function should also be indistinguishable from a pseudorandom function (PRF), and, further, be unforgeable when used as a MAC in the secret key setting, these requirements are formalised by the notions PRF-Pr (PRF Preserving) and MAC-Pr (MAC Preserving), respectively. We also discussed the multi-property-preserving (MPP) paradigm, where hash functions preserve multiple properties simultaneously; we discussed a few examples of constructions from the literature preserving MPP. Finally, we briefly (and informally) discussed how security proofs of hash functions are generally developed in a provable security framework, namely by adopting either the standard model or the ideal model.

Chapter 3

Design of Hash Functions

Recent years have witnessed an exceptional research interest in cryptographic hash functions, especially after the popular attacks against MD5 and SHA-1 in 2005. In 2007, the U.S. National Institute of Standards and Technology (NIST) has also boosted this interest by announcing a public competition to select the next hash function standard, to be named SHA-3. Not surprisingly, the hash function literature has since been rapidly growing in an extremely fast pace. According to [124], around 50-60 hash functions were available in 1993, followed by at least 30-40 others developed since then [126], in addition to the 64 SHA-3 submissions (some of these were standardised in ISO/IEC 10118). In this chapter, we provide a comprehensive, up-to-date discussion of the current state of the art of cryptographic hash functions design. In particular, we present an overview of how (and why) cryptographic hash functions evolved over the years, and then elaborate on some of the most common design approaches. The contents of this chapter was published in [5],[6].

3.1 Introduction

Cryptographic hash functions have indeed proved to be the workhorses of modern cryptography. Their importance was first realised with the invention of public key cryptography (PKC) by Diffie and Hellman [60] in 1976, where it became an integral part of PKC ever since. Unfortunately, recent advances in cryptanalysis revealed inherent weaknesses in most of the popular hash functions triggering an urgent call for further research in this area. In response, two main approaches have been adopted: either *patching* the existing constructions by slightly modifying them to fix a particular

set of weaknesses, or designing new hash functions from scratch. In the first approach, if inherent weaknesses were discovered, they imply that the design principles on which the hash function is based are flawed and unless they are thoroughly revised, it is most likely that those weaknesses will still exist, even if they appear to have been fixed by some minor modifications. Likewise, in the second approach, if a hash function is designed from scratch, it may sufficiently resist a particular set of weaknesses, but may also covertly suffer from other (possibly more severe) weaknesses that might not have been spotted at early development stages. Existing constructions, on the other hand, have the advantage that they have been extensively studied and analysed over time, thus, unless very carefully designed, structurally new hash functions may well be susceptible to more attacks than those that they resist. However, the sponge construction (discussed in section 3.3.3) is an example of a new hash function built from scratch based on totally new design principles, but, at the time of writing, no serious security flaws were reported on the sponge construction.

SHA-3 Competition. For a long time, SHA-1 and MD5 hash functions have been the closest to a hashing de facto, this, however, has changed in 2004 and 2005 when Wang *et al.* [154, 155, 156] showed that finding collisions for MD5 can be easy while substantially reducing the work needed to find collisions on SHA-1 to 2^{69}, which is much less than the expected 2^{80}. Although a break with complexity of 2^{69} is still (at the time of writing) theoretical, it showed that SHA-1 is not as strong and collision-resistant as it is supposed (and thought) to be. This has driven the U.S. National Institute of Standards and Technology (NIST) in November 2007 to announce an open competition[1] to select a new hash functions standard, to be named SHA-3 [117]. NIST received 64 submissions, 51 of which were accepted for round 1 of the competition in December 2008. In July 2009, only 14 round 1 candidates successfully progressed to round 2; these are briefly analysed in [12]. In December 2010, the 5 finalist candidates where chosen (these are, BLAKE, Grøstl, JH, Keccak and Skein), and the winner is expected to be announced in the second quarter of 2012.

Chapter Outline. This chapter is organised as follows. In section 3.2, we classify hash functions as keyed and keyless. Section 3.3 then provides a relatively lengthy and up-to-date discussion about various iterative hash functions, this is indeed the most common approach (at least contemporarily) in designing hash functions. In particular, we discuss the Merkle-Damgård construction (in section 3.3.1), generic attacks against

[1] For a comprehensive resource about SHA-3 competition and all its candidates, see http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo

Merkle-Damgård (in section 3.3.2), and how accordingly the research community tried to patch the construction (in section 3.3.3). Beside iterative functions, tree-based ones have also been proposed, these are briefly discussed in section 3.4. Finally, section 3.5 discusses the most popular approaches of designing compression functions, namely, e.g., based on block/stream-cipher and provably secure hash functions.

3.2 Keyless vs. Keyed Hash Functions

Generally, hash functions are classified as keyless or keyed. Keyless hash functions accept a variable[2] length message M and produce a fixed length hash value, $H : \{0,1\}^* \rightarrow \{0,1\}^n$. Keyed hash functions, on the other hand, accept both a variable length message M and a fixed length key K to produce a fixed length hash value, $H_K : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$. Keyed hash functions can be further classified based on whether the key is private or public. Secretly keyed hash functions are usually used to build Message Authentication Codes (MAC), the canonical example is HMAC [120, 1]; see section 2.5 and chapter 6 for more information about MACs. If, however, the hash functions are publicly keyed, they are commonly known as dedicated-key hash functions [55, 25]. Hash functions designed in the dedicated-key setting are families of hash functions where individual member functions are indexed by different keys. In this setting, if a member of the hash function family was broken, this should have minimal effect on the other members of the same family (this is not the case in the keyless setting where a single attack against a function breaks the function entirely, e.g., [155, 156]). An obvious drawback of hash functions in the dedicated-key setting, however, is a degraded efficiency since in this case the function is required to process an extra input (the key) beside the message input.

In general, a hash function (keyed or keyless) is built out of two components: a compression function f and a construction H. The compression function is a function mapping a larger (but fixed) sized input to a smaller fixed sized output $f : \{1,0\}^m \rightarrow \{1,0\}^n$, where $m > n$. The construction is the way the compression function is repeatedly called to process a message; refer to table 3.1 for brief (informal) definitions of some hash functions terminology which will be used interchangeably throughout the thesis, also see [47] for a discussion about the lack of standard terminology and definitional consistency in the hash functions literature.

[2]The term *variable* in this context indicates that the length of the message is upper bounded by a large number of bits, $M = \{0,1\}^{\leq \lambda}$ (e.g., $\lambda = 64$), that is sufficient to represent any message in practice. The term *arbitrary* [109], on the other hand, describes messages with infinite length, $M = \{0,1\}^*$. However, some authors use the two terms interchangeably. In this thesis, unless stated otherwise, we will always use variable length messages, but for convenience, we will use the notation $\{0,1\}^*$ (that of arbitrary length messages).

Terminology	Informal Definition
Compression/Compressing Function	A standard building block of a hash function, with its domain larger than its range.
Construction, Transform, Mode of Operation, Chaining Mode, Domain Extension Transform, Composition Scheme	An algorithm that systematically makes repeated calls to a building block (often a compression function) to hash a message.
Chaining Variable, Chaining Value, Intermediate Hash, Internal State	The output of a compression function to be used as input to the following compression function call.
Hash value, Hash Code, Hash Result, Hash, Digest, Fingerprint	The final result of hashing a message, which is a fixed length string.

Table 3.1: Hash functions terminology

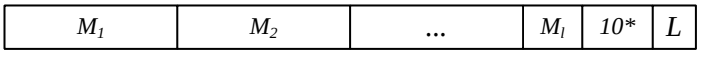
3.3 Iterative Hash Functions

When hash functions first emerged, it was realised that the most convenient way to hash a message is by first dividing it into several blocks and then iteratively and systematically processing these blocks. Today, this sequential hashing approach is still, by far, the most widely used, even with the advent of parallel processors (which, at least in principle, should have given advantage to the parallel hash functions). In the following subsections, we review some popular iterative hashing constructions and discuss how recent designs tried to fix weaknesses in earlier ones. However, note that the absence of a particular construction in the sections below does not imply that we disfavour it; indeed it is nearly impossible to be exhaustive in such a rapidly growing literature.

3.3.1 Merkle-Damgård Construction

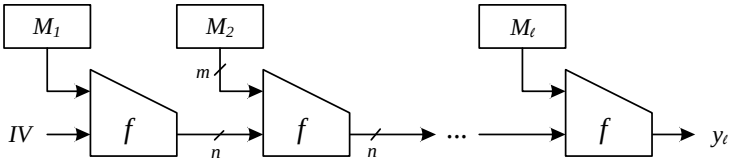
Most of today's popular hash functions, such as MD5 and SHA-1, are based on the infamous Merkle-Damgård construction (also called the cascade construction) proposed independently by Merkle [111] and Damgård [55] in 1989 (though, Damgård's construction was keyed while Merkle's was keyless). However, it appears that similar construction has previously been proposed by Rabin [128] in 1978, raising some controversy in whether it should be called Rabin's construction instead. Nevertheless, while Rabin did indeed propose this construction, it was Merkle and Damgård who formally proved that it is collision resistant if its underlying compression function is collision resistant. In the Merkle-Damgård construction, the message M is first divided into

equally sized blocks, $M = M_1, M_2, \ldots, M_\ell$. If the message M fell over or below the block boundaries, it is padded. To be collision resistance, the length of the message is appended to the message after padding it, this is termed Merkle-Damgård *strengthening* (first coined by Lai and Massey in [96], though already proposed by Merkle [111] and Damgård [55]); figure 3-1 illustrates the padding algorithm[3], where L is a 64-bit encoding of the the length of the message and m is the length of a single block. The message is then iterated repeatedly by calling a Fixed-Input-Length (FIL) compression function $f : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$ accepting two inputs: a message block M_i (of length m) and either an Initialisation Vector IV (when hashing the first block) or a chaining variable (which is the output of the previous f call), both of length n; figure 3-2 provides a depiction and a pseudocode of the Merkle-Damgård construction.



Algorithm $\text{Pad}_s(M)$
$d = M + 1 + 64 \mod m$
$M||1||0^d||\langle M \rangle_{64} \rightarrow \hat{M}$
$\hat{M} \rightarrow M_1 \ldots M_\ell$

Figure 3-1: (Strengthened) Merkle-Damgård padding algorithm



Algorithm MD^f
$M \rightarrow M_1 \ldots M_\ell$
$y_0 = IV$
for $i = 1$ **to** ℓ **do**
  $y_i = f(M_i, y_{i-1})$
return y_ℓ

Figure 3-2: The Merkle-Damgård construction

3.3.2 Generic Attacks Against Merkle-Damgård

Eventually, several weaknesses were found in the Merkle-Damgård construction giving raise to a class of *generic* attacks that is applicable to any hash function based on the plain Merkle-Damgård construction. Note the difference between *generic* and *dedicated* attacks, where dedicated attacks exploit internal structures specific to a particular hash function and thus only affect that hash function (e.g., the attacks against MD4, MD5 and SHA-1 by Wang *et al.* [155, 156] are dedicated attacks). Below we discuss generic attacks against the Merkle-Damgård construction; even though their practical relevance is not clear, they still demonstrate intrinsic structural weaknesses in the construction.

[3]While the padding algorithm illustrated in figure 3-1 is the most commonly used, it is not the only one. The Enveloped constructions such as ESh and CS (section 3.3.3) use different padding algorithms.

The Extension Attack. It is often claimed that this attack was first reported by Ferguson and Schneier [70] in 2003 where it was described as a "surprisingly serious (and simple) flaw" in the Merkle-Damgård construction. However, it seems that the basic idea of this attack was discovered long before 2003 by Solo and Kent who called it the padding attack [151]. We present four variants of this attack as follows:

- *Collision Attack.* Suppose we have a message M with length $|M| = L$ that hashes to $H(M)$, then given any hash function $H(.)$ based on the Merkle-Damgård construction, a collision is trivially found as follows: $H(M||pad||x) = H(H(M)||x)$ where pad is the padding appended to the message M before being hashed and $|pad| = L \bmod m$, where m is the length of a single block in M, which is usually $|H(M)|$; note that $|pad|$ can indeed be 0 if the message was perfectly aligned at block boundaries. However, this attack does not consider Merkle-Damgård strengthening (appending the message length to the message before hashing it).

- *Second Collision Attack.* In this attack, a collision can easily be found by extending equally sized already colliding messages. That is, if we have $H(M) = H(N)$ while $M \neq N$ and $|M| = |N|$, a second collision can be obtained by extending M and N with an arbitrary string (suffix) S, $H(M||S) = H(N||S)$. This will work with Merkle-Damgård strengthening, but without strengthening, a second collision is even easier as the colliding messages no longer have to be equally sized.

- *Related Message Attack.* With Merkle-Damgård, one can easily compute a related/extended message M' for an unknown message M by only knowing L (length of M) and $H(M)$, that is, $H(M||L||x)$ is the hash of a message consisting of the original M and extended by a suffix $L||x$; again, since the attacker knows L, it is trivial to figure out how M has been padded before being hashed. This attack indeed does not affect collision resistance, but it shows that the Merkle-Damgård construction does not behave like a random oracle, which is a desirable property that hash functions should possess; see section 2.3 and chapter 5.

- *MAC Forgery Attack [48].* In this attack, one can compute (forge) a valid message without knowing the secret key K used by a MAC algorithm based on Merkle-Damgård that generates a tag T from M. Suppose we have a MAC algorithm that processes a message using a Merkle-Damgård based hash function H by prepending its key to the message $\text{MAC}(K, M) = H(K||M)$. Now, one can update the message M by appending a suffix Y and obtain a tag T' matching the new message $M||Y$ without even knowing K, i.e., $\text{MAC}(K, M||Y) = H(K||M||Y)$.

The Multi-collision Attack. In [89], Joux showed that finding multiple collisions (more than two messages hashing to the same value) in a Merkle-Damgård hash function is not much harder than finding single collisions. In his multi-collision attack, Joux assumed access to a machine $\dot{C}$ that given an initial state, returns two colliding messages ($\dot{C}$ may use the birthday attack or any other attack exploiting weaknesses in the corresponding hash function). Figure 3-3 illustrates the attack.

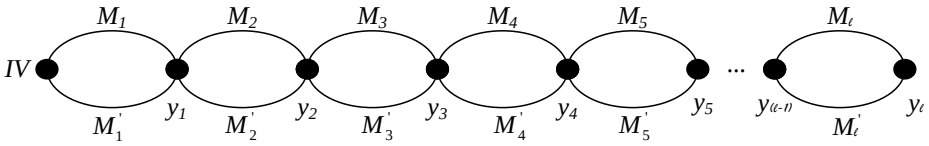


Figure 3-3: Multi-collision attack

In figure 3-3, initially, the IV is sent to $\dot{C}$ which returns M_1 and M_1' colliding at y_1, then y_1, in turn, is sent to $\dot{C}$ which returns M_2 and M_2' colliding at y_2, this process continues until reaching y_ℓ. It is easy to see that any combination of the messages preceding y_ℓ will collide in y_ℓ. In fact, in the example in figure 3-3, there are 2^ℓ messages all colliding in y_ℓ and the cost of generating these collisions is only ℓ times the cost of generating single collisions (as generated by machine $\dot{C}$).

2nd Pre-image Attack. This attack was first proposed by Dean in [58] and later generalised by Kelsey and Schneier in [92]. The attack assumes the existence of a set of expandable messages; these are messages of different lengths but produce the same intermediate hash value (chaining variable) given a particular IV. Expandable messages, however, do not produce the same final hash value due to Merkle-Damgård strengthening. These messages can easily be found if the hash function contains fixed points[4]. While not an intrinsic weakness of the Merkle-Damgård construction, fixed points can be found in many Merkle-Damgård implementations (e.g., SHA-1) because the compression functions are usually modelled as Davies-Meyer functions where the chaining variable input of the compression function is further XORed with its output. Let H be a Merkle-Damgård hash function, and f be its compression function. Suppose we have a set of such messages $E = E_1, \ldots, E_l$ which all produce an intermediate hash value y_E. Let $M = M_1, M_2 \ldots, M_\ell$ be a very long message consisting of ℓ blocks, and $C = c_1, \ldots, c_{\ell-1}$ be the set of all the intermediate hash values of M (for a message consisting of ℓ blocks, there are $\ell - 1$ intermediate hash values). Now, search for a block M_i in M, such that $f(y_E, M_i) \in C$. Suppose M_i is found (finding M_i has complexity

[4] A fixed point is found when two consecutive chaining values collide, that is $f(h_{i-1}, M_i) = h_i = h_{i-1}$, where f is a compression function.

less than 2^n since M is a very long message, where n is the length of the final hash value of H) and it matches c_j (the j-th intermediate hash value of M), now search E (the set of expandable messages) for a message E_s of length j-1 such that the number of blocks of $E_s||M_i$ is j. Let the original message M without its first j blocks be M', then $H(E_s||M_i||M') = H(M)$. This attack finds a 2nd pre-image for a message of size 2^k in $2^{n/2+1} + 2^{n-k+1}$ steps rather than the expected 2^n. For example, using RIPEMD-160, it finds a 2nd pre-image for a 2^{60} byte message in around 2^{106} steps, rather than the expected 2^{160} steps, where RIPEMD-160 produces a hash digest of size 160 bits [92].

The Herding Attack. This attack is due to Kelsey and Kohno [91] and is closely related to the multi-collision and 2nd pre-image attacks discussed above. A typical scenario where this attack can be used is when an adversary commits to a hash value D (which is not random) that he makes public and claims (falsely) that he possesses knowledge of unknown events (events in the future) and that D is the hash of that knowledge. Later, when the corresponding events occur, the adversary tries to herd the (now publicly known) knowledge of those events to hash to D as he previously claimed. The attack is based on a diamond structure and proceeds in two main phases.

- phase 1: construct the diamond and calculate the value D.
- phase 2: given a prefix, find a suffix and herd it to D through the diamond.

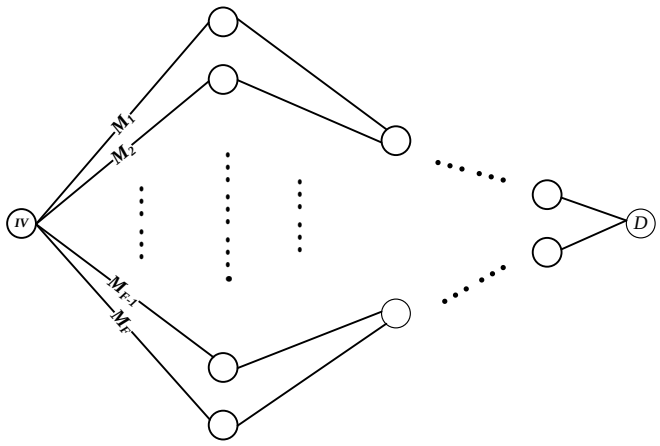


Figure 3-4: Diamond structure

In phase 1, the attacker constructs a diamond structure as shown in figure 3-4, where the vertices are hash values and the edges are messages. If two messages meet in a vertex, they collide at that vertex. Initially, the attacker randomly generates an arbitrary large number of initial messages, $M_1, \ldots, M_F$, hashes them and tries to find

collisions, then repeats until reaching the root of the diamond, D. Once the diamond is constructed, any path from the initial messages to D will hash to D. In phase 2, the attacker *herds* a given prefix P to hash to D as follows: first, the attacker searches for a suitable 1-block suffix S that if concatenated with P, it will produce a hash colliding with one of the hash values of the initial messages $H(M_i)$ where $i \in \{1, 2, \ldots, F\}$; for 2^k initial messages, 2^{n-k} trials are required to find such a suffix (where n is the length of the final hash). Once a match is found, P, S and the sequence of messages from the matching $H(M_i)$ to D are concatenated, and this whole string will eventually hash to D. The herding attack was recently extended to non-Merkle-Damgård constructions [67, 10]. A more detailed complexity analysis of the herding and diamond-based attacks are presented in [148], which points out a flaw in the construction proposed in [91] to produce a diamond structure, and provide computational complexity analysis for constructing the diamond. To resist this attack, hash functions should possess the Chosen Target Forced Prefix (CTFP) pre-image resistance property; see section 2.2.4.

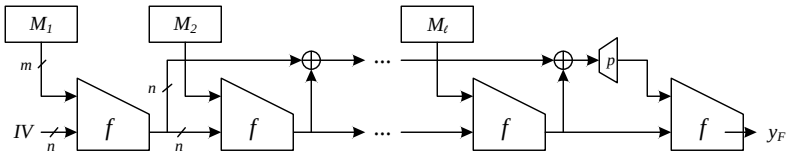
3.3.3 Variants of Merkle-Damgård

The discovery of the weaknesses reported in section 3.3.2 drove the research community to propose modified variants of the Merkle-Damgård construction that patch such weaknesses. In this section, we present a few examples of both keyless and keyed constructions (most of which trade off efficiency for security). Note that some of these constructions use different padding algorithms than the standard one in figure 3-1.

Wide and Double Pipe. One of the earliest proposals to enhance the Merkle-Damgård construction is the wide/double pipe by Lucks [101] who showed that increasing the size of the internal state (i.e., the chaining variable) to become larger than the size of the final hash value, would significantly improve the security of the hash function. This modification clearly thwarts the extension attack since in the wide/double pipe the final hash value is truncated, so in order to append an extension, the unknown discarded bits have to be guessed, which is clearly difficult if the number of the discarded bits is non-trivial. Furthermore, by increasing the size of the internal state, finding collisions for the compression function becomes harder, which complicates the other generic attacks. An obvious drawback of the wide/double pipe, however, is a degraded efficiency as the compression function now has larger input/output while keeping the hashing rate constant (length of message block is fixed) since the chaining variable input is increased. Also, adapting existing hash functions for the wide/double pipe may be difficult since it might be the only reasonable way to increase the internal state is to use multiple compression function calls in parallel for every iteration. Re-

cently, Yasuda [161] adopted a slightly modified variant of the double pipe construction and proved its unforgeability beyond the birthday barrier.

The 3C Construction. Another variant of the Merkle-Damgård construction is the 3C construction [75] which basically maintains a variable containing a value produced by repeatedly XORing the chaining variables while hashing a message; this variable is then processed in an extra finalisation call to the compression function. Figure 3-5 illustrates the 3C construction (where P is a padding function). An enhanced variant of 3C is 3C+ which uses extra memory, but makes finding *multi-block* collisions more difficult (not to be confused with multi-collision attack, see section 2.2). However, in [88], it was shown that both 3C and 3C+ are indeed susceptible for multi-block attack; this was demonstrated using a recent attack against MD5 that was found to be applicable for both the plain Merkle-Damgård and 3C/3C+. Furthermore, 3C does not resist multi-collision, 2nd pre-image and herding attacks [73, 74].

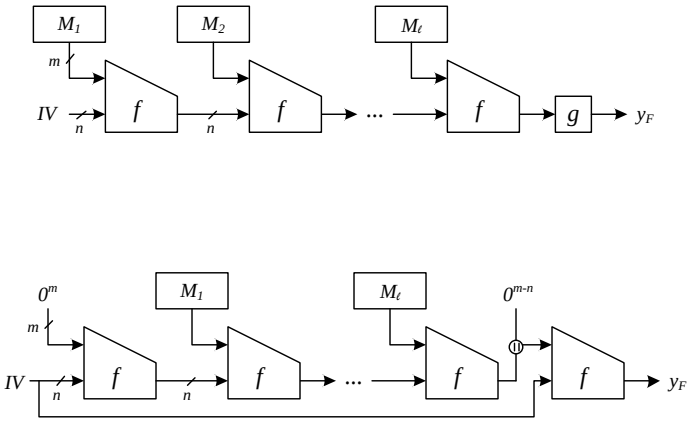


Algorithm $3C^f$
$M \rightarrow M_1 \dots M_\ell$
$y_0 = IV, t = 0$
for $i = 1$ **to** ℓ **do**
  $y_i = f(M_i, y_{i-1})$
  $t = t \oplus y_i$
return
  $y_F = f(P(t), y_\ell)$

Figure 3-5: The 3C construction

The Prefix Free, Chop, NMAC and HMAC Constructions. Several constructions were proposed by Coron *et al.* in [48] as immediate fixes to the Merkle-Damgård construction after showing that the latter is not indifferentiable from $\mathcal{RO}$ (see section 2.3 for details about the indifferentiability notion). However, Bellare and Ristenpart [24] later showed that even though these constructions are indifferentiable from $\mathcal{RO}$, they are not collision resistant. The prefix-free construction does not modify the Merkle-Damgård construction, instead it modifies the padding algorithm to make sure that the message is prefix free. One way to do this is by prepending or appending the length of the whole message to every message block. However, beside wasting a few bits to represent the length of the message in every block and so degrading the efficiency, this obviously does not work well with streaming applications (where the length of the message is not known beforehand). The chop construction basically removes a non-trivial number of bits from the final hash value. This, while it solves the

indifferentiability issue, unfortunately lowers the security bounds of the hash function. In NMAC, an independent function g is applied to the output of the last application of the compression function, while in HMAC an extra compression function call is introduced. The NMAC and HMAC constructions proposed by Coron *et al.* in [48] are not to be confused with the popular NMAC/HMAC [1, 21] developed as MACs. Coron's NMAC and HMAC constructions are illustrated in figure 3-6.



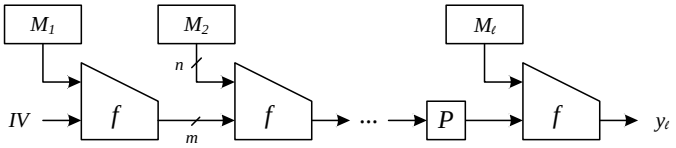
Algorithm NMAC^{f,g}
$M \rightarrow M_1 \dots M_\ell$
$y_0 = IV$
for $i = 1$ **to** ℓ **do**
  $y_i = f(M_i, y_{i-1})$
return $y_F = g(y_\ell)$

Algorithm HMAC^{f}
$M \rightarrow M_1 \dots M_\ell$
$M_0 = 0^m, y_0 = f(M_0, IV)$
for $i = 1$ **to** ℓ **do**
  $y_i = f(M_i, y_{i-1})$
return
  $y_F = f(y_\ell || 0^{m-n}, IV)$

Figure 3-6: The NMAC and HMAC constructions

The Merkle-Damgård with Permutation. In [81] Hirose *et al.* proposed the Merkle-Damgård with Permutation (MDP) construction which introduces very minor modification to the plain Merkle-Damgård. The only difference between the plain Merkle-Damgård and MDP is that in MDP the chaining variable input of the last compression function is permuted. The authors proved that MDP is indifferentiable from $\mathcal{RO}$ while the collision resistance of MDP follows trivially from the collision resistance of the Merkle-Damgård construction as the former introduces minimal changes to the latter. The authors also discussed the security of possible simple MAC constructions based on MDP. However, although with such a simple modification, the authors succeeded in proving a significant security gain, MDP seems to be able to thwart only the extension attack, but not other Merkle-Damgård generic attacks. Also, recently it was shown that MDP is neither pre-image nor 2nd pre-image resistant [11]. Figure 3-7 illustrates MDP, where $\pi(.)$ is a permutation function.

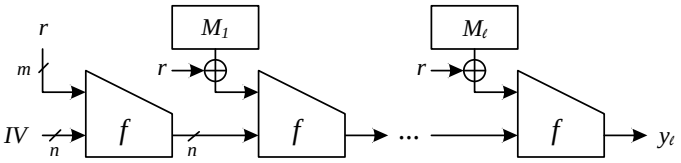
Randomized Hashing. Randomized hashing [78] is not quite a variant of Merkle-Damgård, rather it is a generic fix that can be applied to any construction (including



Algorithm MDP^f
$M \to M_1 \dots M_\ell$
$y_0 = IV$
for $i = 1$ **to** $\ell - 1$ **do**
  $y_i = f(M_i, y_{i-1})$
return $y_\ell = f(\pi(y_{\ell-1}), M_\ell)$

Figure 3-7: The MDP (Merkle-Damgård with Permutation) construction

Merkle-Damgård). In randomized hashing the input of the hash function is randomised using a *salt*, leaving the construction unmodified; figure 3-8 illustrates the RMX transform [79], which is an instantiation of the randomized hashing paradigm. The authors claim that randomized hashing will strengthen any hash function, even the weakest ones. Randomized hashing was originally proposed for digital signatures where a message M is first randomised with a salt r to produce a randomised message M'. A digital signature sig is then generated from M'. The original message M, the salt r and the signature sig are then sent to the verifier. When the verifier receives these parameters, it first randomises M with r to produce M' and carries out standard signature verification using M' and sig. Randomized hashing for digital signatures are standardised by NIST in [57].

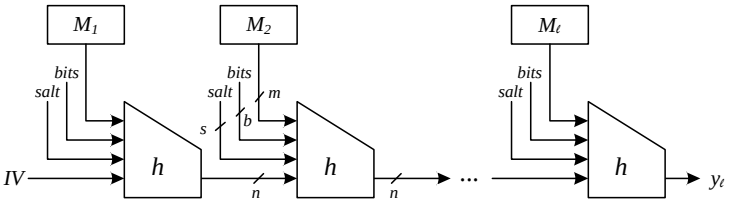


Algorithm RMX_r^f
$M \to M_1 \dots M_\ell$
$y_0 = f(r, IV)$
for $i = 1$ **to** ℓ **do**
  $y_i = f(M_i \oplus r, y_{i-1})$
return y_ℓ

Figure 3-8: The RMX construction

HAIFA Framework. HAsh Iterative FrAmework (HAIFA) is a modified Merkle-Damgård construction proposed by Dunkelman and Biham [66], see figure 3-9 for an illustration. HAIFA modifies Merkle-Damgård by introducing extra input parameters to the compression function. These are: a *salt* value (used as a key to create families of hash functions—if only one hash function is needed, the salt is set to 0), and the number of *bits* hashed so far, which thwarts many of the generic attacks against the plain Merkle-Damgård construction since the input to every compression function call becomes (with high probability) unique and highly dependent on where the compression function call is made through the hashing chain. In fact, HAIFA can be considered a dedicated-key hash function [25]. The idea of adding additional input parameters to

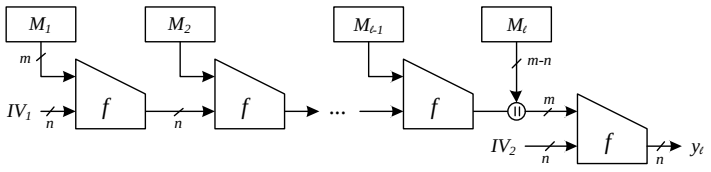
the compression function has been previously proposed by Rivest through a process called *dithering* [132]; though a 2nd pre-image attack against dithered hash functions was reported by Andreeva *et al.* in [9]. An obvious drawback of HAIFA is efficiency degradation since the compression function now has more input parameters to process. Furthermore, HAIFA cannot be (easily) used to patch existing Merkle-Damgård based hash functions because a compression function designed for the Merkle-Damgård construction would not naturally accommodate the extra HAIFA parameter inputs.



Algorithm HAIFA^h_s
$M \rightarrow M_1 \dots M_\ell$
$y_0 = IV$
for $i = 1$ **to** ℓ **do**
$y_i = h(M_i, y_{i-1}, b_i, s)$
return y_ℓ

Figure 3-9: The HAIFA framework

Enveloped Merkle-Damgård. The Enveloped Merkle-Damgård (EMD) construction was proposed in [24] by Bellare and Ristenpart when they were introducing their multi-property-preserving notion, where they recommend that a particular hashing scheme should preserve multiple properties at the same time. This stemmed from the fact that they were able to prove the four constructions proposed by Coron *et al.* in [48] are not collision resistant while still being indifferentiable from $\mathcal{RO}$. Bellare and Ristenpart showed that EMD preserves collision resistance, indifferentiability from $\mathcal{RO}$ and indistinguishability from Pseudorandom Function (PRF). Figure 3-10 illustrates EMD.

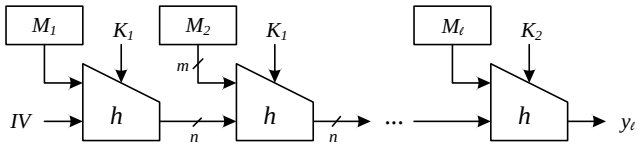


Algorithm $\text{EMD}^f_{IV_1,IV_2}$
$M \rightarrow M_1 \dots M_\ell$
$y_0 = IV_1$
for $i = 1$ **to** $\ell - 1$ **do**
$y_i = f(M_i, y_{i-1})$
return
$y_\ell = f(y_{i-\ell} || M_\ell, IV_2)$

Figure 3-10: The EMD (Enveloped Merkle-Damgård) construction

Nested Iteration. An and Bellare proposed the Nested Iteration (NI) mode of operation while they were proving that the Merkle-Damgård construction can be used to construct a Variable-Input-Length (VIL) MAC from a Fixed Input Length (FIL)

MAC. NI is basically a keyed variant of the Merkle-Damgård construction making use of two keys $k_1, k_2 \in \{0,1\}^k$. Figure 3-11 illustrates the NI construction. Beside being unforgeable, Bellare and Ristenpart later proved in [25] that NI is also indistinguishable from PRF, indifferentiable from $\mathcal{RO}$, and if strengthening was used, NI is also collision resistant. However, neither NI nor its strengthened variant is target collision resistant (TCR); see section 2.7 for discussion about the TCR property.

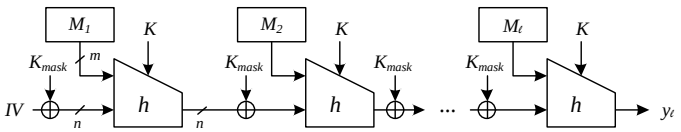


Algorithm $\mathrm{NI}^h_{K_1,K_2}$

$M \rightarrow M_1 \ldots M_\ell$
$y_0 = IV$
for $i = 1$ **to** $\ell - 1$ **do**
  $y_i = h(K_1, M_i, y_{i-1})$
return $y_\ell = h(K_2, M_\ell, y_{\ell-1})$

Figure 3-11: The NI (Nested Iteration) construction

Shoup (Sh) Construction. In [141], Shoup proposed an elegant keyed construction. In addition to the key input of the compression function, the chaining variables of every compression function iteration in Sh is further XORed with a key mask; figure 3-12 illustrates the Sh construction. A variant of the Sh construction has been proposed by Bellare and Ristenpart in [25] that makes the last compression function call a *wrapping* call (this last application of the compression function is called an *envelop*). Thus, this variant is called the Envelop Shop (ESh), which has been proven to preserve five important properties, namely: collision resistance, unforgeability, indifferentiability from $\mathcal{RO}$, indistinguishability from PRF and TCR. In [129] Reyhanitabar *et al.* further showed that ESh preserves the ePre property but not Sec, aSec, Pre and aPre (for information about these properties, see section 2.7 and [135]).



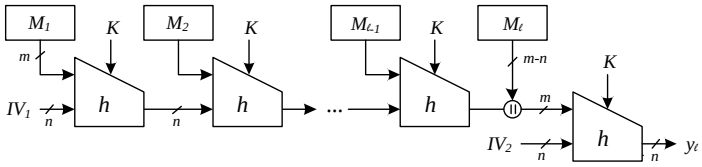
Algorithm $\mathrm{Sh}^h_{K,K^{mask}}$

$M \rightarrow M_1 \ldots M_\ell$
$y_0 = IV$
for $i = 1$ **to** ℓ **do**
  $y_i = h(y_{i-1} \oplus K^{mask}, M_i)$
return y_ℓ

Figure 3-12: The Shoup construction

Chaining Shift. The Chaining Shift (CS) construction CS: $\{0,1\}^k \times \{0,1\}^{n+n} \times \{0,1\}^* \rightarrow \{0,1\}^n$ was proposed by Maurer and Sjödin in [109] as a more efficient solution than the NI construction for constructing AIL-MAC from FIL-MAC; figure 3-13

depicts the CS construction, which uses a FIL compression function $f : \{0,1\}^{m+n} \rightarrow \{0,1\}^n$. The CS construction was shown to be unforgeable [109], indistinguishable from PRF [25], indifferentiable from $\mathcal{RO}$ [24], and the strengthened variant of it (with strengthened padding) is collision resistant. Maurer and Sjödin have also simultaneously proposed the Chaining Rotate (CR) construction, which is similar to CS.



Algorithm $\mathrm{CS}^h_{K,IV_1,IV_2}$

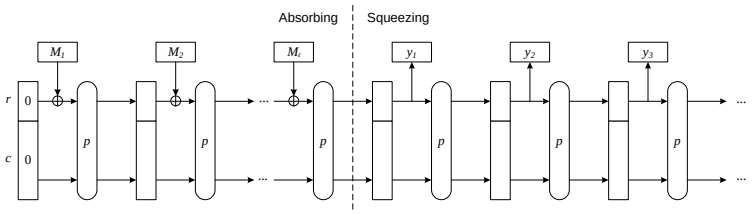
$M \rightarrow M_1 \ldots M_\ell$
$y_0 = IV$
for $i = 1$ **to** $\ell - 1$ **do**
  $y_i = h(y_{i-1}, M_i)$
return
  $y_\ell = h(IV_2 || y_{\ell-1}, M_\ell)$

Figure 3-13: The CS (Chaining Shift) construction

3.3.4 Sponge Construction

Based on totally different design principles than Merkle-Damgård's, the Sponge construction is a newly proposed and promising hashing construction [33]. Basically, sponge hashing proceeds in two phases, the absorbing phase and the squeezing phase (and hence its name). The sponge operates on a fixed-length state $b = \{0,1\}^{r+c}$, composed of r bits (called bit-rate) and c bits (called capacity), through a function $p : \{0,1\}^{r+c} \rightarrow \{0,1\}^{r+c}$ which produces a transformation or permutation of b. In the absorbing phase, the message is divided into r-bit blocks (padded if necessary) and each block is XORed with the r part of b (initially, $b = 0^{r+c}$), p then iteratively processes b until all blocks are exhausted. In the squeezing phase, the state continues to be transformed/permuted by p but this time the r parts of the states are returned at every iteration as output blocks. Since the sponge construction supports variable length output, the user chooses the length of the final hash value which determines how many of the returned blocks in the squeezing phase need to be returned. Figure 3-14 illustrates the sponge construction. An example of a hash function based on the sponge construction is Keccak [35] which has recently been selected (along with 4 others) to advance to the final stage of the SHA-3 competition (see section 3.1). Recently, Andreeva *et al.* introduced a generalisation of the sponge functions, which they call "The Parazoa Family" [13].

Although still considered an iterative construction, the sponge is completely different from the Merkle-Damgård construction; which obviously means that the generic attacks discussed in section 3.3.2 are not applicable. Moreover, the sponge construc-



Algorithm $Spng_n^p$
$M \rightarrow M_1 \dots M_\ell$
$r = 0, c = 0$
for $i = 1$ **to** ℓ **do**
  $p(r \oplus M_i, c) = (r, c)$
for $i = 1$ **to** n **do**
  $Y = Y||r$
  $p(r, c)$
return Y

Figure 3-14: The Sponge construction

tion has been proven to be indifferentiable from $\mathcal{RO}$ in [34]. However, that does not mean that the sponge construction is not susceptible to other kinds of attacks, it is just that (at the time of writing) such attacks have not been discovered yet. Recently, Gorski *et al.* [77] showed that hash functions based on the sponge framework may be susceptible to slide attack. An obvious disadvantage of sponges is that their relatively large state slows down the full diffusion of bits, hence, the sponge construction may be more suitable for hashing large messages.

3.4 Tree-based Hash Functions

Figure 3-15 illustrates a typical tree-based hashing construction. This is the most parallelisable class of constructions and is mainly suited for multi-core platforms where multiple processors can independently operate on different parts of the message simultaneously. An early tree-based mode of operation was proposed by Damgård [55] which was later optimised by Sarkar and Scellenberg [140], and Pal and Sarkar [122]. Similarly, Bellare and Rogaway [27] used a tree-based approach to build UOWHFs [116] which are weaker than collision-resistant hash functions. More recent works for building tree-based UOWHFs include [99] and [139]. In [23] Bellare and Micciancio proposed the *randomize-and-combine* paradigm, where the message is split into blocks, randomised individually and then combined by an operation such as XOR (though XOR-based combiners are broken by a linear algebra attack on long messages [103]). Although this structure was originally proposed to build *incremental* functions[5], it can be thought of as a 2-level tree and can still be parallelised since the randomisation process of the individual blocks are independent (i.e., can be carried out by different threads/processors). Tree-based constructions are slightly less popular than the iter-

[5] An incremental function can efficiently *update* the digest of a previously hashed message to reflect any changes without having to re-hash the whole message.

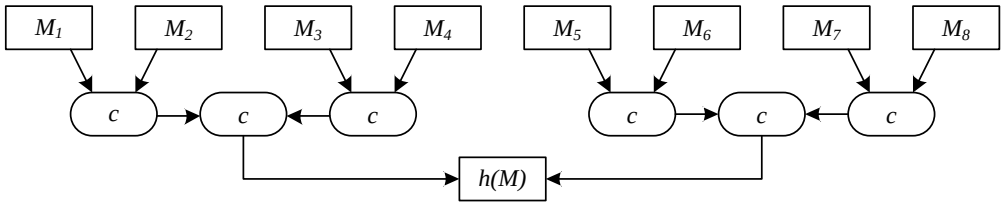


Figure 3-15: Sample tree construction

ative ones because they are not as suitable for low-end platforms such as smart cards and RFID, which limits their utility. Skein [69] and MD6 [133] hash functions (SHA-3 candidates) provide a tree hashing mode beside the conventional iterative mode.

3.5 Compression Functions

Some hash functions are built from scratch such as MD5 and SHA-1, these are called dedicated hash functions. Others are built based on existing cryptographic or mathematical components that were not originally designed to be used for hashing but could be *tailored* to. So far, we have not explicitly discussed how to construct compression functions because when we design a hash function, one would assume the presence of a "good" compression function and design a construction accordingly. This obviously becomes an issue in practice. Thus, some proposals were exclusively concerned with building a compression function that preserves some property X and then adopt a suitable construction that is provably preserves X if the underlying compression function also preserves X (e.g., the Merkle-Damgård construction is collision resistant if the underlying compression function is also collision resistant).

3.5.1 Hash Functions Based on Block and Stream Ciphers

Building hash functions based on block ciphers is the most popular and established approach. In this approach, the compression function is a block-cipher with its two inputs representing a message block and a key. Preneel, Govaerts and Vandewalle [127] studied the 64 possible ways of constructing hash functions from a block-cipher $E : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$. These 64 constructions are sometimes called PGV constructions after the authors' initials who used an attack-based[6] analysis approach to study the security of these constructions. It was then reported that 12 out of the 64 PGV constructions are collision resistant, but later Black *et al.* [38] showed (using proof-based approach this time) that another 8 PGV constructions are also collision

[6]Constructions that resisted the authors' attacks were deemed secure.

resistant if they were properly iterated, even if their underlying compression functions are not collision resistant. The most widely adopted construction of these 20 PGV construction is the one attributed to Davies and Meyer [110]: $y_i = f(y_{i-1}, M_i) \oplus y_{i-1}$, where y_{i-1} and M_i are the input of the compression function f, and y_i is its output. Another popular PGV construction is the Matyas-Meyer-Oseas construction [38], which is the opposite of the Davies-Meyer one. In Matyas-Meyer-Oseas, the output of the compression function y_i is further XORed with the message block input M_i (rather than the chaining variable y_{i-1} in Davies-Meyer). Further analyses and proofs of the collision resistance and pre-image resistance of these PGV constructions in the ideal cipher model can be found in [68] and [144]. Although PGV functions are provably secure, they are inefficient because the key (which represents the message block input of the compression function) is changed with every compression function call, and this is undesirable with block-ciphers since changing the key rapidly requires huge amount of computation (due to key setup). Thus, another approach is to use fixed-key block-cipher based compression functions [37, 136, 142, 143]. In this approach, a small non-empty set of keys are fixed and used for the block-cipher (when called by the compression function), while wrapping the block-cipher with other arbitrary functions to process the other compression function's input that was previously used as a key. However, Black *et al.* [37] proved that such construction, making a single call to the fixed-key block-cipher, although efficient, cannot be collision resistant.

An inherent problem with designing hash functions based on block-ciphers is that block ciphers usually have small block size (e.g., 128 bit) which is insufficient to maintain an acceptable hash function security[7], unless the result of the hash function can be expanded, which proved to be even more difficult. A particularly interesting solution to this dilemma is designing double block length (DBL) compression functions where the compression function outputs double the size of the underlying block-cipher [115, 80]. Clearly, however, DBL based hash functions still scarify some efficiency.

Although the stream-cipher-based approach is less popular than the block-cipher one, in the recent SHA-3 competition, some of the successful second round candidates were based on stream-ciphers (e.g., CubeHash [31]). The main differences between block-cipher and stream-cipher based hash functions are the block size and the number of rounds. In block-cipher-based, message blocks are usually large, and iterated a small number of rounds, while in stream-cipher-based, message blocks are small, with more rounds. Thus, in block-cipher-based, a good compression function is necessary but in stream-cipher-based, even a weak compression function may provide sufficient security.

[7] For a block size of 128 bit, a collision can be found in 2^{64} operations due to the birthday attack, which may be at the edge of feasibility with the recent supercomputers processing technologies.

3.5.2 Hash Functions Based on Mathematical Problems

The majority of today's well known hash functions process the message by mixing its bits in such a way that fulfils the various security and statistical requirements, but their security cannot usually be mathematically proven since they are not based on mathematical models. Provably secure cryptographic hash functions, on the other hand, is a class of hash functions that are based on mathematical problems where a rigorous mathematical proof can be derived such that it reduces breaking these hash functions to finding a solution to some hard mathematical problems. Examples of such hash functions include: hash functions based on the discrete logarithm problem [39, 45], hash functions based on the factorisation problem [46], hah functions based on finding cycles in expander graphs [44], etc. Designing several cryptographic primitives, including hash functions, based on the Knapsack problem[8] was more popular during the 90's. Although these schemes had good software and hardware performance, most of them are broken [125] which made knapsack based design approach less attractive. An example of such hash functions is the one proposed by Damgård [55] based on additive knapsack which was cryptanalysed in [123], and the LPS hash function [44] based on a multiplicative knapsack, which was cryptanalysed in [149].

Another example of provably secure hash functions is syndrome-based hash functions [17, 18] which are based on an NP-complete problem known as *Regular Syndrome Decoding*. However, even though syndrome-based hash functions are provably secure, cryptanalytic results were published against several versions of these functions [49, 138, 72]. A recent syndrome based hash function is FSB [16] which was submitted to the SHA-3 competition, but failed to progress to round 2 of the competition, mostly due to its slow performance and excessive memory consumption, which seems to be the case with most hash functions based on mathematical problems (e.g., while the authors of DAKOTA [56] claim that it performs better than many other modular arithmetic-based hash functions, it is still approximately 18 times slower than SHA-256). Recently, Bernstein *et al.* [32] proposed an enhanced variant of FSB, called RFSB (Really fast FSB), and claimed that it runs at 13.62 cycles/byte, which is faster than SHA-2.

3.5.3 Other Approaches

Occasionally, attempts were made to adopt less common approaches when designing hash functions, most of which have not attracted much interest. In this section, we discuss two such approaches: chaos-based and cellular automata based hash functions.

[8] Knapsack is a famous combinatorics problem, requiring the optimised selection of items $d_1, \ldots, d_n$, where each item has a value v_i and a weight w_i, such that the sum of the values $\sum_{i=1}^{n} v_i$ is maximised while keeping the sum of the weights $\sum_{i=1}^{n} w_i$ less than a particular threshold T.

Chaos-based Hash Functions. Chaos theory is the mathematical representation of dynamic systems. These systems possess many desirable properties that suit the requirements of hash functions. For example, chaotic systems are very sensitive to changes in their initial values, potentially fulfiling the desirable hash function property requiring the output of the hash function to be highly sensitive to changes in its input; this phenomena is called the *avalanche effect* (also called butterfly effect in the chaos theory literature). Moreover, chaotic systems are one way functions and unpredictable. Hash functions based on chaos theory use *chaotic maps*, which are functions that exhibit particular chaotic behaviours; examples of these maps include: logistic map [104], tent map [162], and cat map [59]. Unfortunately, most chaos-based hash functions suffer from poor efficiency due to their inherent complex structure, which makes them unattractive as a practical approach for building hash functions.

Cellular Automata-based Hash Functions. Cellular Automata (CA) are discrete time models consisting of collections of cells organised in a grid, and each cell has a current state. The states of the cells evolve over time depending on their current states and the states of the neighbouring cells. CA were originally used by von Neumann [153] while he was studying self-reproducing systems and then popularised by Wolfram's substantial work in this area [158] who observed that based on simple rules, very complex behaviours can be obtained. Damgård was the first to propose a hash function based on CA [55], but his proposal was cryptanalysed by Daemen *et al.* [51] who, in turn, proposed another CA-based hash function, called CellHash. The same authors later proposed SubHash [52] which is an improved version of CellHash; both CellHash and SubHash are hardware-oriented and were cryptanalysed in [42]. Another hash function based on CA was proposed by Mihaljevie *et al.* [112].

3.6 Summary

In this chapter we provided a thorough discussion of the state of art of hash functions design. Roughly speaking, hash functions can either be keyless or keyed. Each class has different applications and is based on different design principles. Hash functions can also be classified as iterative or parallel. While iterative functions are indeed the most common, parallelisable hash functions are increasingly being popularised with the rapid advent of parallel systems. We provided a lengthy discussion about the popular Merkle-Damgård construction, how it fell prey to various generic attacks and what modifications were proposed to patch it. Finally, we also discussed how compression functions are being designed and what approaches are adopted.

Chapter 4

Integrated-Key Hash Functions

Traditionally, hash functions were designed in the keyless setting, where a hash function accepts a variable-length message and returns a fixed-length fingerprint. Unfortunately, over the years, significant weaknesses were reported on instances of some popular keyless hash functions. This has motivated the research community to start considering the dedicated-key setting, which also allows for more rigorous security arguments. However, it turns out that converting an existing keyless hash function into a dedicated-key one is non-trivial since the keyless compression function does not normally accommodate the extra key input. In this chapter we formalise an approach that can potentially solve this problem. In this approach, keyless hash functions are seamlessly transformed into keyed variants by introducing an extra component accompanying the (still keyless) compression function to handle the key separately. Hash functions constructed in this setting are called integrated-key hash functions. We propose several integrated-key constructions and prove their collision, pre-image and 2nd pre-image resistance.

4.1 Introduction

Recent years have witnessed a research proliferation in the cryptographic hash functions. Hash functions have traditionally been designed in the *keyless* setting where a construction (iteration mode) $H^f : \mathcal{M} \rightarrow \{0,1\}^n$ with access to a keyless compression function $f : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$ hashes a variable-length message $M \in \mathcal{M}$ by iteratively calling f. However, as a result of the recent attacks [154, 155, 156] against several widely used keyless hash functions, such as MD5 and SHA-1, another approach has increasingly been popularised. In that approach hash functions are constructed in the

dedicated-key setting [25], where a family of hash functions $C^h : \mathcal{K} \times \mathcal{M} \rightarrow \{0,1\}^n$, with access to a *publicly keyed* compression function $h : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$, is constructed, such that instances (members) of C^h are indexed by different public keys $k_i \in \mathcal{K}$. Hash functions constructed in the dedicated-key setting are not to be confused with secretly keyed hash functions usually used to build Message Authentication Codes (MACs); these are discussed extensively in chapter 6.

Dedicated-Key. In [25], Bellare and Ristenpart discussed in length the benefits of the dedicated-key setting and showed how adopting this setting potentially improves hash function heterogeneity, which enables users to utilise different instances of the same hash function family. That is, if an attack against an instance (member) of the hash function family (specified by a particular key) was found, then it only breaks that particular instance while very likely having almost negligible (or even no) effect on other instances (indexed by different keys). The dedicated-key setting also improves the security guarantees of hash functions and gives an easy solution to Rogaway's *foundation of hashing dilemma* [134], which states that keyless hash functions cannot be collision resistant due to the pigeonhole principle (see section 2.1) where he solved it by means of explicit (but slightly complex) problem reduction. Instead, the dedicated-key setting allows for simpler and more straightforward theoretical arguments about collision resistance of hash functions. An obvious drawback of the dedicated-key setting is slight efficiency loss due to the introduction of an extra input (i.e., the key) that the hash function needs to process beside the message input, but such efficiency loss seems to be unavoidable in any keyed setting.

Integrated-Key. Unfortunately, in most cases, existing keyless hash functions cannot be easily turned into dedicated-key hash functions[1] because their corresponding keyless compression functions do not naturally accommodate the key input introduced in the dedicated-key setting. This means that the dedicated-key approach may, in most cases, only be used in designing new hash functions, not to patch/strengthen existing ones. Thus, in this chapter we try to answer the following question:

> *Given a keyless hash function $H^f : \mathcal{M} \rightarrow \{0,1\}^n$, where $f : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$, how to construct a keyed hash function $C : \mathcal{K} \times \mathcal{M} \rightarrow \{0,1\}^n$ from H^f without modifying the internal structure of f, while still keying the compression function as in the dedicated-key setting?*

[1] MAC schemes (e.g., HMAC) can trivially turn keyless hash functions into keyed ones, but these schemes are secretly keyed, while here we are interested in the dedicated-key setting, where keys are publicly-known.

While here we explicitly require keyless hash functions to be keyed as in the dedicated-key setting (where the key is processed with every application of the compression function), there are other approaches that can key a hash function without strictly adhering to this requirement, which makes adapting a keyless hash function to accept a key input easy and efficient. So, why do we insist on matching the dedicated-key setting, while there are other cheaper and more convenient approaches? To answer this question, let's consider an example where a keyless hash function is keyed by processing the key only at the last compression function call (e.g., EMD [24]). Now suppose that an attack exploiting the intermediate chaining variables was found, this should readily break the whole function family because apart from the last compression function call, the whole hashing procedure is common for all members. Clearly, this is not the case with the dedicated-key hash functions. Therefore, in this chapter, we introduce a *bridging* approach that seamlessly transforms a keyless hash function into a dedicated-key one without modifying the internal (keyless) structure of the former, we call hash functions constructed in this setting *integrated-key hash functions*.

Related Work. There is a distinction between merely creating families of hash functions and creating them in the dedicated-key setting, where the latter explicitly states that all applications of the compression function should be keyed. Examples of constructions forming families of hash functions, but without keying all compression function applications, are EMD (Enveloped Merkle-Damgård) [24], and BCM (Backward Chaining Mode) [15]. EMD was originally proposed as a keyless hash function introducing a second IV which can be used as a key. On the other hand, BCM was explicitly proposed as a way of creating families of hash functions out of keyless compression functions, but it still does not strictly follow the dedicated-key approach since the keys are not applied to every applications of the compression function. A proposal that can be seen as a variant of an integrated-key construction is RMX [79] which is based on the randomized hashing paradigm [78] (see section 3.3.3 for detailed description of the RMX construction). RMX performs some sort of key whitening (a block-cipher technique) by combining a random salt (i.e., a key) with every message block before sending it to the compression function. However, randomized hashing is more suitable for digital signatures where the hash function is used by the signature algorithm as a black-box. That is, since the message M in RMX is merely XORed with the *publicly available* key K, it is possible to retrieve parts of M via some differential and linear cryptanalysis techniques [36, 106], but if RMX is used in digital signatures, the signature algorithm will further process the hash value produced by RMX, potentially resisting differential/linear cryptanalysis. Other recent hash functions such as Skein [69] and MD6 [133]

support both keyless and keyed modes, but to the best of our knowledge, there is currently no formal work considering integrated-key-like constructions (creating families of dedicated-key hash functions out of keyless primitives).

Chapter Outline. The rest of the chapter is organised as follows. In section 4.2 we introduce and formally define the integrated-key approach of designing hash functions and discuss both its advantages and drawbacks. In section 4.3 we proposed several Merkle-Damgård-based integrated-key constructions, and then proceed in section 4.4 to prove that these constructions are collision resistant (in section 4.4.1), 2nd pre-image resistant (in section 4.4.2) and pre-image resistant (in section 4.4.3).

4.2 Integrated-key Hash Functions

In the integrated-key setting, we start with a keyless hash function H^f and convert it to a dedicated-key hash function[2] $\hat{C}^{f,g}$ by leaving the (keyless) compression function f *untouched* and introducing a lightweight mixing function g that handles the extra key input independently outside f, we call g the *integration function*. In particular, the integrated-key approach creates families of hash functions $\hat{C}^{f,g}_{\mathcal{K}} = \{\hat{C}^{f,g}_{k_1}, \hat{C}^{f,g}_{k_2}, \ldots, \hat{C}^{f,g}_{k_n}\}$ out of keyless hash functions, where each member is indexed by a different key $k_i \in \mathcal{K}$. Formally, the integrated-key hash functions are defined as follows:

Definition 4.2.1 (Integrated-key hash functions). *A hash function family $\hat{C}^{f,g} : \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$, with access to a keyless compression function $f : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^n$, and a keyed integration function $g : \{0,1\}^k \times \{0,1\}^b \to \{0,1\}^b$ with $b \in \{n, m\}$, is said to be in the integrated-key setting if given a variable-length message $M \in \mathcal{M}$ and a fixed-length key $k \in \mathcal{K}$, M is iteratively processed by f in fixed-length blocks, while k is processed by g at least with every application of f.*

In the integrated-key setting, both the compression function and the integration function can still be treated as a single composite component and thus simulating the dedicated-key setting. Indeed, in chapter 6 we formally show that the integrated-key variant of a particular hash function is indistinguishable from its dedicated-key variant, except with negligible probability. We emphasise that the integrated-key approach creates families of hash functions from keyless hash functions by only changing the way the key input is handled, it will not, however, strengthen or remedy any construction-

[2] We denote a dedicated key hash function by $C^{\cdot}$ (with access to a keyed compression function), while its integrated-key variant is denoted by $\hat{C}^{\cdot,\cdot}$ (with access to both a keyless compression function and an keyed integration function).

based weakness of a keyless hash function should its dedicated-key variant[3] be already vulnerable to such weakness. That is, if a particular keyless hash function is susceptible to some attacks and those attacks can be thwarted if we transform that function to a dedicated-key one, then transforming it to an integrated-key hash function will still thwart those attacks. However, if such attacks cannot be thwarted at the dedicated-key setting, they will not be thwarted at the integrated-key setting. This obviously holds because the two settings are indistinguishable as we will show in chapter 6.

Key Generation and Usage. Like the dedicated-key setting, keys in the integrated-key setting are fixed, meaning that once a key is specified, the same key is repeatedly used at every hashing iteration; using more than one key to hash a single message implies that more than one members of the hash function family are used to hash a message, which is an uncommon practice and clearly further degrades the efficiency. We also require the key to be random. In fact, in section 6.2.2, we show that if keys are generated randomly, integrated-key hash functions are indistinguishable from Pseudo-random Functions (PRF), where the latter is an idealisation of a hash function, see section 2.4 for information about PRF. Finally, we assume that the keys are generated by honest parties, but this is not a major concern since most practical hash functions applications, such as digital signatures, message authentication codes (MAC) etc., only require security for honestly generated keys [25].

Efficiency. Obviously, the integrated-key approach is less efficient than the keyless one, increasing the computational overhead of hashing a message M by at least $\ell \cdot \tau_g$, where τ_g is the cost of the integration function g, and ℓ is the number of message blocks of M. The efficiency of an integrated-key hash function highly depends on the implementation of g, which ideally should be a lightweight (but sufficiently mixing) function. In the dedicated-key setting, on the other hand, the mere introduction of the key increases the input space of the compression function so even with a lighter implementation of the compression function (which may trade off security for efficiency), the function still needs to deal with a larger input; see appendix A for a comprehensive investigations in how to optimise the efficiency of hash functions implementations.

Compatibility. The main motivation for introducing the integrated-key setting is to conveniently convert a keyless hash function to a dedicated-key one. In this sense,

[3]We obtain the dedicated-key variant of a keyless hash function by modifying the latter's keyless compression function to accommodate the key input introduced in the dedicated-key setting (a process that is usually expensive and non-trivial in practice).

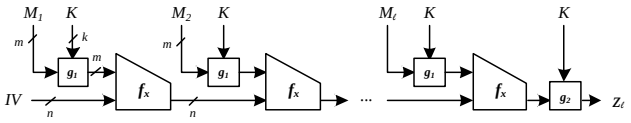
the integrated-key setting can be considered a variant of the (more general) dedicated-key setting. Thus, the two settings are clearly compatible (you can convert one to another if required). This is also immediate from our indistinguishability result in section 6.2.1 where we showed that an adversary cannot distinguish between the two settings. In fact, in an abstract level, both the compression and integration functions at the integrated-key setting can be seen as a single compressing entity accepting a key input, which resembles a conventional dedicated-key compression function (i.e., thus, abstractly, the dedicated-key and integrated-key settings are equivalent).

4.3 The iMD Constructions

In this section, we propose three integrated-key constructions based on the popular (strengthened) Merkle-Damgård construction [111, 55], namely[4] x-iMD (input-based iMD), y-iMD (chaining-based iMD), c-iMD (output-based iMD), which we collectively called the "iMD constructions". Figure 4-1 illustrates the constructions graphically and provides pseudocode. Before initiating the hashing process, the message is padded by the padding function $Pad_s(.)$, which appends a 1 bit followed by 0^r bits, such that $r + 1 + \langle M \rangle_\beta \mod d = 0$, where $\langle M \rangle_\beta$ is a β-bit encoding of the message length $|M|$ (the most common value of β is 64); this algorithm is illustrated in figure 3-1. In x-iMD and y-iMD, the message block M_i, respectively the chaining variable y_{i-1}, is processed by the integration function before calling the compression function. In contrast, c-iMD calls the integration function only after calling the compression function. In all the constructions, the compression functions $f_x, f_y, f_c : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$, and the integration functions $g_1 : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^m, g_2 : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ are treated as black-boxes that are called iteratively by the constructions while hashing the message. All constructions additionally apply an extra application of g_2 before returning the final hash value; as will be shown in section 4.4, this "enveloping" call hides the internal structure of the schemes and thus improves their security. Our iMD constructions bear some resemblances to the NMAC construction proposed in [48] at the keyless setting.

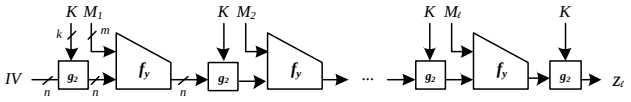
The integration function can be *any* lightweight sufficiently mixing function. We suggest adopting the prepend-permute-chop (PPC) paradigm, which was proven indifferentiable from $\mathcal{RO}$ in [133, 63]. The integration function can then be defined as follows: $g(K, c) = [\pi(K||c)]^n$, where π is a random permutation and $[X]^n$ is the most significant n-bits of X. A practical example of a function adopting the PPC approach

[4] The names of the constructions are derived from the notation used in representing their inputs/outputs and indicate where the integration function is placed around the compression function.



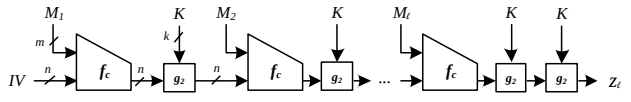
Algorithm x-MD$^{f_x,g_1,g_2}(M)$

$M_1 ... M_\ell \leftarrow Pad_s(M)$
$y_0 = IV$
for $i = 1$ **to** ℓ **do**
  $y_i = f_x(g_1(K, M_i), y_{i-1})$
return $z_\ell = g_2(K, y_\ell)$



Algorithm y-MD$^{f_y,g_2}(M)$

$M_1 ... M_\ell \leftarrow Pad_s(M)$
$y_0 = IV$
for $i = 1$ **to** ℓ **do**
  $y_i = f_y(M_i, g_2(K, y_{i-1}))$
return $z_\ell = g_2(K, y_\ell)$



Algorithm c-MD$^{f_c,g_2}(M)$

$M_1 ... M_\ell \leftarrow Pad_s(M)$
$y_0 = IV$
for $i = 1$ **to** ℓ **do**
  $y_i = g_2(K, f_c(M_i, y_{i-1}))$
return $z_\ell = g_2(K, y_\ell)$

Figure 4-1: Graphical representation and pseudocode of the iMD constructions

is the compression function of MD6 [133], where the random permutation π is represented by a series of operations, but unlike the MD6 compression function, for the sake of our integration function, these operations should be sufficiently light to maintain acceptable overall hashing efficiency. There might be other potential candidates for the integration function, but we do not discuss the design of the integration function any further; in chapter 7, we list this as an extension to the thesis.

4.4 Security Analysis

In the following subsections we prove that x-iMD, y-iMD, c-iMD are collision resistant (CR), pre-image resistant (Pre) and 2nd pre-image resistant (Sec). The CR and the Sec proofs are provided in the standard model, while the Pre proof is provided in the Random Oracle Model (ROM). The proofs of this chapter adopt similar approaches to the ones in [14, 15, 109] while being conducted in the integrated-key setting. In the following proofs we adopt the standard cryptographic proof approach which proves that a construction preserves a particular property given that its underlying primitives already preserve this property (see section 2.8 for details about cryptographic proof

approaches). However, it can be argued that one of the goals of the integrated-key setting is to strengthen a weak compression function, potentially contradicting our proof approach (i.e., we prove that a construction preserves xxx, given its compression function preserves xxx, but we know that the compression function does not preserve xxx). This means that we are making slightly strong assumptions on the compression functions, which is the case, except that while strengthening hash functions is one motivation of using the integrated-key approach, it is not the only nor the main one. Indeed, the Integrated-key setting is mainly used to convert keyless hash functions into dedicated-key ones for applications requiring the latter (not necessarily to strengthen the former). As discussed in section 4.2, the integrated-key setting will only strengthen a function as much as its dedicated-key variant does, meaning that the integrated-key variant of a keyless function will only provide as much security as what its dedicated-key variant can provide, but not better. Therefore, to simplify the proof, we will follow the general standard proof approach and assume that both the compression and integration functions preserve xxx to prove that the iMD constructions also preserve xxx.

4.4.1 Collision Resistance (CR)

One can argue that since the iMD constructions introduce minor modifications to the (strengthened) Merkle-Damgård construction, and the latter is collision resistant [55], then the iMD constructions are also collision resistant. In this section we formalise this argument. That is, given a key K, we prove that $F(M) \neq F(M')$ holds as long as $M \neq M'$, except with negligible probability, where $F \in \{x\text{-iMD}, y\text{-iMD}, c\text{-iMD}\}$.

Theorem 4.4.1 (CR of x-iMD, y-iMD, c-iMD). *Let the keyless compression functions* $f_x : f_y : f_c : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$ *be* $(t_{f_x}, m+n, \epsilon_{f_x})$-*cr*, $(t_{f_y}, m+n, \epsilon_{f_y})$-*cr*, $(t_{f_c}, m+n, \epsilon_{f_c})$-*cr, and let the keyed integration functions* $g_1 : \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m, g_2 : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ *be* $(t_{g_1}, m+k, \epsilon_{g_1})$-*cr*, $(t_{g_2}, n+k, \epsilon_{g_2})$-*cr. Then,* $x\text{-iMD}^{f_x,g_1,g_2}$, $y\text{-iMD}^{f_y,g_2}$, $c\text{-iMD}^{f_c,g_2}$ *accessing* $\{f_x, g_1, g_2\}, \{f_y, g_2\}, \{f_c, g_2\}$, *respectively, are* (t_x, L, ϵ_x)-*cr*, (t_y, L, ϵ_y)-*cr*, (t_c, L, ϵ_c)-*cr, such that:*

1. *for* x-*iMD:* $\epsilon_x \leq \epsilon_{f_x} + \epsilon_{g_2} + \epsilon_{g_1}$, *and* $t_x \leq \frac{(t_{f_x}+t_{g_2}+t_{g_1})-3\ell\cdot\tau_{f_x}\tau_{g_1}-\tau_{g_2}}{3}$
2. *for* y-*iMD:* $\epsilon_y \leq 2\epsilon_{g_2} + \epsilon_{f_y}$, *and* $t_y \leq \frac{(t_{f_y}+2t_{g_2})-\tau_{g_2}(3\ell\cdot\tau_{f_y}+1)}{3}$
3. *for* c-*iMD:* $\epsilon_c \leq 2\epsilon_{g_2} + \epsilon_{f_c}$, *and* $t_c \leq \frac{(t_{f_c}+2t_{g_2})-\tau_{g_2}(3\ell\cdot\tau_{f_c}+1)}{3}$

where $\tau_{f_x}, \tau_{f_y}, \tau_{f_c}$ *are the costs of calling the compression functions* f_x, f_y, f_c, *and* τ_{g_1}, τ_{g_2} *are the costs of calling the integration functions* g_1 *and* g_2, *respectively,* $L = max\{|M|, |M'|\}$, $\ell = \lceil (L+1+\beta)/m \rceil$, β *is the length of the encoding of* L, *and* m *is the length of a single message block.*

Proof. Since the iMD constructions use strengthening (appending the length of the message before hashing it), we have $\text{iMD}(K, M) \neq \text{iMD}(K, M')$ trivially holds as long as $|M| \neq |M'|$ for any iMD construction. Thus, here we will only consider the case when $|M| = |M'|$. Let A_x, A_y, A_c be (t_x, L, ϵ_x)-cr, (t_y, L, ϵ_y)-cr, (t_c, L, ϵ_c)-cr adversaries against x-iMD, y-iMD, c-iMD, and let $B_{1,x}, B_{2,x}, B_{3,x}, B_{1,y}, B_{2,y}, B_{3,y}, B_{1,c}, B_{2,c}, B_{3,c}$ be $(t_{B_{1,x}}, m+n, \epsilon_{B_{1,x}})$-cr, $(t_{B_{2,x}}, k+n, \epsilon_{B_{2,x}})$-cr, $(t_{B_{3,x}}, k+m, \epsilon_{B_{3,x}})$-cr, $(t_{B_{1,y}}, m+n, \epsilon_{B_{1,y}})$-cr, $(t_{B_{2,y}}, k+n, \epsilon_{B_{2,y}})$-cr, $(t_{B_{3,y}}, k+n, \epsilon_{B_{3,y}})$-cr, $(t_{B_{1,c}}, m+n, \epsilon_{B_{1,c}})$-cr, $(t_{B_{2,c}}, k+n, \epsilon_{B_{2,c}})$-cr, $(t_{B_{3,c}}, k+n, \epsilon_{B_{3,c}})$-cr, adversaries against the compression functions f_x, f_y, f_c and the integration functions g_1, g_2, as defined in figures 4-2, 4-3, 4-4. In the x-iMD, y-iMD, c-iMD games, the adversaries $\{B_{1,x}, B_{2,x}, B_{3,x}\}, \{B_{1,y}, B_{2,y}, B_{3,y}\}, \{B_{1,c}, B_{2,c}, B_{3,c}\}$ run the adversaries A_x, A_y, A_c, respectively, as follows:

- adversaries $B_{1,x}, B_{1,y}, B_{1,c}$ attack f_x, f_y, f_c to find internal collisions,
- adversaries $B_{2,x}, B_{2,y}, B_{2,c}$ attack the last (randomising) application of the integration function g_2 to find a collision in the final hash value,
- adversary $B_{3,x}$ attack the iterative internal calls of g_1 in the x-iMD game, while adversaries $B_{3,y}, B_{3,c}$ attack the iterative internal calls of g_2 in the y-iMD and the c-iMD games ($B_{3,y}$ and $B_{3,c}$ are identical except that one of them attack the y-iMD game while the other attack the c-iMD game).

Throughout the games, adversaries A_x, A_y, A_c output message pairs $M \neq M'$ which are then parsed by adversaries $B_{1,x}, B_{2,x}, B_{3,x}, B_{1,y}, B_{2,y}, B_{3,y}, B_{1,c}, B_{2,c}, B_{3,c}$ into blocks and hashed iteratively. The probabilities that a collision is found in those games are then the probabilities that adversaries A_x, A_y, A_c succeed in outputting collisions. Conversely, if adversaries A_x, A_y, A_c found a collision, this implies that at least one of the adversaries $B_{1,x}, B_{2,x}, B_{3,x}, B_{1,y}, B_{2,y}, B_{3,y}, B_{1,c}, B_{2,c}, B_{3,c}$ succeeded in their collision games. That is, generally, collisions in the iMD constructions must be the result of collisions in the internal compression function, the internal integration function, or the last finalisation call of the integration function. This holds as long as we make the explicit assumption that the output of all functions are uniformly distributed, which is a common assumption in most cryptographic proofs (see section 1.3), especially that the output of one function is usually fed to the input of another.

In the x-iMD game, the probability that $B_{1,x}$ and $B_{2,x}$ will succeed are the collision probabilities of f_x and g_2 respectively, that is $\mathbf{Adv}^{\text{cr}}_{B_{1,x}} = \epsilon_{f_x}$ and $\mathbf{Adv}^{\text{cr}}_{B_{2,x}} = \epsilon_{g_2}$. Similarly, the probability that $B_{3,x}$ will succeed is the collision probability of the internal

Adversary $B_{1,x}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_x(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = f_x(g_{1,K}(x_i), y_{i-1})$
$z_i = y_i$
$z'_i = f_x(g_{1,K}(x'_i), y'_{i-1})$
$z'_i = y'_i$
if $z_i = z'_i \wedge (g_{1,K}(x_i), y_{i-1}) \neq (g_{1,K}(x'_i), y'_{i-1})$;
then $\alpha = i$; **break;**
end for
if $\alpha = 0$
then return $\perp$
else return
$(x_\alpha, y_{\alpha-1}), (x'_\alpha, y'_{\alpha-1})$

Adversary $B_{2,x}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_x(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = f_x(g_{1,K}(x_i), y_{i-1})$
$z_i = y_i$
$z'_i = f_x(g_{1,K}(x'_i), y'_{i-1})$
$z'_i = y'_i$
end for
if $g_{2,K}(z_\ell) = g_{2,K}(z'_\ell) \wedge z_i \neq z'_i$
then $\alpha = 1$
if $\alpha = 0$,
then return $\perp$
else return
$(x_\ell, y_{\ell-1}), (x'_\ell, y'_{i-\ell})$

Adversary $B_{3,x}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_x(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = f_x(g_{1,K}(x_i), y_{i-1})$
$z_i = y_i$
$z'_i = f_x(g_{1,K}(x'_i), y'_{i-1})$
$z'_i = y'_i$
if $g_{1,K}(x_i) = g_{1,K}(x'_i) \wedge x_i \neq x'_i$
then $\alpha = i$; **break;**
end for
if $\alpha = 0$,
then return $\perp$
else return
$(x_\alpha, y_{\alpha-1}), (x'_\alpha, y'_{\alpha-1})$

Figure 4-2: Adversaries $B_{1,x}, B_{2,x}, B_{3,x}$ in the CR game

Adversary $B_{1,y}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_y(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = f_y(x_i, g_{2,K}(y_{i-1}))$
$z_i = y_i$
$z'_i = f_y(x'_i, g_{2,K}(y'_{i-1}))$
$z'_i = y'_i$
if $z_i = z'_i \wedge (g_{2,K}(x_i), y_{i-1}) \wedge (g_{2,K}(x'_i), y'_{i-1})$;
then $\alpha = i$; **break;**
end for
if $\alpha = 0$,
then return $\perp$
else return
$(x_\alpha, y_{\alpha-1}), (x'_\alpha, y'_{\alpha-1})$

Adversary $B_{2,y}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_y(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = f_y(x_i, g_{2,K}(y_{i-1}))$
$z_i = y_i$
$z'_i = f_y(x'_i, g_{2,K}(y'_{i-1}))$
$z'_i = y'_i$
end for
if $g_{2,K}(z_\ell) = g_{2,K}(z'_\ell) \wedge z_i \neq z'_i$
then $\alpha = 1$
if $\alpha = 0$
then return $\perp$
else return
$(x_\ell, y_{\ell-1}), (x'_\ell, y'_{\ell-1})$

Adversary $B_{3,y}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_y(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = f_y(x_i, g_{2,K}(y_{i-1}))$
$z_i = y_i$
$z'_i = f_y(x'_i, g_{2,K}(y'_{i-1}))$
$z'_i = y'_i$
if $g_{2,K}(y_{i-1}) = g_{2,K}(y'_{i-1}) \wedge x_i \neq x'_i$
then $\alpha = i$; **break;**
end for
if $\alpha = 0$
then return $\perp$
else return
$(x_\alpha, y_{\alpha-1}), (x'_\alpha, y'_{\alpha-1})$

Figure 4-3: Adversaries $B_{1,y}, B_{2,y}, B_{3,y}$ in the CR game

Adversary $B_{1,c}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_c(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = g_{2,K}(f_c(x_i, y_{i-1}))$
$z_i = y_i$
$z'_i = g_{2,K}(f_c(x'_i, y'_{i-1}))$
$z'_i = y'_i$
if $(x_i, y_{i-1}) \neq (x'_i, y'_{i-1}) \wedge$
$f_c(x_i, y_{i-1}) = f_c(x'_i, y'_{i-1})$
then $\alpha = i$; **break;**
end for
if $\alpha = 0$
then return $\perp$
else return
$(x_\alpha, y_{\alpha-1}), (x'_\alpha, y'_{\alpha-1})$

Adversary $B_{2,c}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_c(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = g_{2,K}(f_c(x_i, y_{i-1}))$
$z_i = y_i$
$z'_i = g_{2,K}(f_c(x'_i, y'_{i-1}))$
$z'_i = y'_i$
end for
if $g_{2,K}(z_\ell) = g_{2,K}(z'_\ell)$
$\wedge z_i \neq z'_i$
then $\alpha = 1$
if $\alpha = 0$
then return $\perp$
else return
$(x_\ell, y_{\ell-1}), (x'_\ell, y'_{\ell-1})$

Adversary $B_{3,c}$
$K \xleftarrow{\$} \mathcal{K}; \alpha = 0$
$\{M, M'\} \leftarrow A_c(K)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M)$
$x_1, \ldots, x_\ell \leftarrow Pad_s(M')$
$y_0 = y'_0 = IV$
for $i = 1$ **to** $i = \ell$ **do**
$z_i = g_{2,K}(f_c(x_i, y_{i-1}))$
$z_i = y_i$
$z'_i = g_{2,K}(f_c(x'_i, y'_{i-1}))$
$z'_i = y'_i$
if $f_c(x_i, y_{i-1}) \neq f_c(x'_i, y'_{i-1})$
$\wedge z_i = z'_i$; **then** $\alpha = i$
break;
end for
if $\alpha = 0$
then return $\perp$
else return
$(x_\alpha, y_{\alpha-1}), (x'_\alpha, y'_{\alpha-1})$

Figure 4-4: Adversaries $B_{1,c}, B_{2,c}, B_{3,c}$ in the CR game

integration function g_1, that is $\mathbf{Adv}^{\text{cr}}_{B_{3,x}} = \epsilon_{g_1}$. Succinctly,

$$\begin{aligned}
\mathbf{Pr}[\text{Coll}_{A_x}] &= \mathbf{Pr}[\text{Coll}_{B_{1,x}}] + \mathbf{Pr}[\text{Coll}_{B_{2,x}}] + \mathbf{Pr}[\text{Coll}_{B_{3,x}}] \\
\mathbf{Adv}^{\text{cr}}_{A_x} &= \mathbf{Adv}^{\text{cr}}_{B_{1,x}} + \mathbf{Adv}^{\text{cr}}_{B_{2,x}} + \mathbf{Adv}^{\text{cr}}_{B_{3,x}} \\
&= \mathbf{Adv}^{\text{cr}}_{f_x} + \mathbf{Adv}^{\text{cr}}_{g_2} + \mathbf{Adv}^{\text{cr}}_{g_1} \\
&\leq \epsilon_{f_x} + \epsilon_{g_2} + \epsilon_{g_1}
\end{aligned}$$

The success probabilities of $B_{1,y}, B_{2,y}, B_{3,y}$ in the y-iMD game are similar to those in the x-iMD game above, except that in y-iMD the internal integration function is g_2 instead of g_1, that is $\mathbf{Adv}^{\text{cr}}_{B_{3,y}} = \epsilon_{g_2}$. Succinctly,

$$\begin{aligned}
\mathbf{Pr}[\text{Coll}_{A_y}] &= \mathbf{Pr}[\text{Coll}_{B_{1,y}}] + \mathbf{Pr}[\text{Coll}_{B_{2,y}}] + \mathbf{Pr}[\text{Coll}_{B_{3,y}}] \\
\mathbf{Adv}^{\text{cr}}_{A_y} &= \mathbf{Adv}^{\text{cr}}_{B_{1,y}} + \mathbf{Adv}^{\text{cr}}_{B_{2,y}} + \mathbf{Adv}^{\text{cr}}_{B_{3,y}} \\
&= \mathbf{Adv}^{\text{cr}}_{f_y} + \mathbf{Adv}^{\text{cr}}_{g_2} + \mathbf{Adv}^{\text{cr}}_{g_2} \\
&\leq \epsilon_{f_y} + \epsilon_{g_2} + \epsilon_{g_2} \leq 2\epsilon_{g_2} + \epsilon_{f_y}
\end{aligned}$$

In the c-iMD game, the integration function g_2 is located behind the compression function f_c, so for $B_{1,c}$, if a collision is found in f_c, it will propagate through g_2 and will

generate an internal collision, this happens with probability $\mathbf{Adv}^{\mathrm{cr}}_{B_{1,c}} = \epsilon_{f_c}$. Similarly, it is easy to see that $B_{2,c}$ and $B_{3,c}$ succeed with probability ϵ_{g_2} each. Succinctly,

$$\begin{aligned}
\mathbf{Pr}[\mathrm{Coll}_{A_c}] &= \mathbf{Pr}[\mathrm{Coll}_{B_{1,c}}] + \mathbf{Pr}[\mathrm{Coll}_{B_{2,c}}] + \mathbf{Pr}[\mathrm{Coll}_{B_{3,c}}] \\
\mathbf{Adv}^{\mathrm{cr}}_{A_c} &= \mathbf{Adv}^{\mathrm{cr}}_{B_{1,c}} + \mathbf{Adv}^{\mathrm{cr}}_{B_{2,c}} + \mathbf{Adv}^{\mathrm{cr}}_{B_{3,c}} \\
&= \mathbf{Adv}^{\mathrm{cr}}_{f_c} + \mathbf{Adv}^{\mathrm{cr}}_{g_2} + \mathbf{Adv}^{\mathrm{cr}}_{g_2} \\
&\leq \epsilon_{f_c} + \epsilon_{g_2} + \epsilon_{g_2} \leq 2\epsilon_{g_2} + \epsilon_{f_c}
\end{aligned}$$

The running time t_x of x-iMD is the running time of the adversary A_x (which outputs the collision). The latter is calculated based on the running times of $B_{1,x}, B_{2,x}, B_{3,x}$, which call adversary A_x as part of their execution, thus, intuitively, the running time of A_x is less than the running times of $B_{1,x}, B_{2,x}, B_{3,x}$, where the running time of the latter is the time of hashing the messages M, M'. However, occasionally, $B_{1,x}, B_{2,x}, B_{3,x}$ will terminate before hashing the whole message (if they find a collision). Therefore, the running time of $B_{1,x}, B_{2,x}, B_{3,x}$ is $B_{1,x} + B_{2,x} + B_{3,x} \leq 3(\tau_{A_x} + \ell(\tau_{f_x}\tau_{g_1})) + \tau_{g_2}$, where $\tau_{f_x}, \tau_{g_1}, \tau_{g_2}$ are the costs of running f_x, g_1, g_2, respectively, and $\ell = \lceil (L+1+\beta)/m \rceil$ such that $L = max\{|M|, |M'|\}$, β is the length of the encoding representing L in bits and m is the length of a single message block[5]. We multiple $\tau_{f_x}\tau_{g_1}$ by 3 because all adversaries execute f_x and g_1 iteratively, but add one instance of ϵ_{g_2} because only adversary $B_{2,x}$ executes the finalisation call of g_2. Finally, solving for τ_{A_x}, we get:

$$\tau_{A_x} \leq \frac{(B_{1,x} + B_{2,x} + B_{3,x}) - 3\ell \cdot \tau_{f_x}\tau_{g_1} - \tau_{g_2}}{3}$$

The running times of τ_{A_y}, τ_{A_c} in the y-iMD, c-iMD games are calculated similarly.

$$\begin{aligned}
\tau_{A_y} &\leq \frac{(B_{1,y} + B_{2,y} + B_{3,y}) - 3\ell \cdot \tau_{f_y}\tau_{g_2} - \tau_{g_2}}{3} \\
&\leq \frac{(B_{1,y} + B_{2,y} + B_{3,y}) - \tau_{g_2}(3\ell \cdot \tau_{f_y} - 1)}{3}
\end{aligned}$$

$$\begin{aligned}
\tau_{A_c} &\leq \frac{(B_{1,c} + B_{2,c} + B_{3,c}) - 3\ell \cdot \tau_{f_c}\tau_{g_2} - \tau_{g_2}}{3} \\
&\leq \frac{(B_{1,c} + B_{2,c} + B_{3,c}) - \tau_{g_2}(3\ell \cdot \tau_{f_c} - 1)}{3}
\end{aligned}$$

This completes the proof. □

[5] We add an extra 1 bit to L and β to adhere to the padding rules.

4.4.2 2nd Pre-image Resistance (Sec)

2nd pre-image resistance is directly implied by collision resistance (every collision resistant hash function is also 2nd pre-image resistant) [135]. This holds if we assumed that the underlying compression function is collision resistant. However, collision resistance is a much stronger assumption than 2nd pre-image resistance, and since a central goal in cryptography is to develop proofs based on as weak assumptions as possible, in this section we prove that the iMD constructions are 2nd pre-image resistant assuming that the compression/integration functions possess weaker-than collision resistance properties. That is, given a key K and a message M, we prove that an adversary cannot find a 2nd message M' such that $F(K, M) = F(K, M')$, except with negligible probability, where $F \in \{x\text{-iMD}, y\text{-iMD}, c\text{-iMD}\}$. One may think that since our iMD constructions are variants of the Merkle-Damgård construction, and the keyed Merkle-Damgård construction is not 2nd pre-image resistant [14], then this implies that our iMD constructions are not 2nd pre-image resistant too (as the two are indistinguishable due to the result in chapter 6). We show that this is not the case. While we consider our iMD constructions close variants of the Merkle-Damgård construction, they are not identical to it due to the addition of a finalisation call to the integration function g_2 in the iMD constructions. However, for technical reasons (which we will discuss later) we were still unable to prove the Sec security of our iMD constructions based on the assumption that the compression functions are only 2nd pre-image resistant (Sec). Instead, we assume that the compression functions possess a property that is weaker than collision resistance (but slightly stronger than Sec), namely that the compression functions are fixed suffix 2nd pre-image resistant (s-Sec); we still assume, nevertheless, that the integration functions are (the weaker) 2nd pre-image resistant (Sec). We first define the s-Sec notion.

Definition 4.4.2 (Fixed Suffix 2nd Pre-image (s-Sec)). *A function F is said to be s-Sec if given a message $M||S$ and its hash value $F(M||S)$, it is infeasible for any (computationally bounded) adversary A to find a 2nd message $M'||S$ such that $F(M||S) = F(M'||S)$ while $M' \neq M$ and S is an arbitrary suffix string of fixed length.*

We can now proceed to prove that the x-iMD, y-iMD, c-iMD constructions are Sec secure given that their compression functions are s-Sec secure and their integration functions are Sec secure.

Theorem 4.4.3 (Sec of x-iMD, y-iMD, c-iMD). *Let the keyless compression functions $f_x : f_y : f_c : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$ be $(t_{f_x}, m+n, \epsilon_{f_x})$-s-sec, $(t_{f_y}, m+n, \epsilon_{f_y})$-s-sec, $(t_{f_c}, m+n, \epsilon_{f_c})$-s-sec, and the keyed integration functions $g_1 : \{0,1\}^k \times$*

$\{0,1\}^m \rightarrow \{0,1\}^m, g_2 : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ *be* $(t_{g_1}, m+k, \epsilon_{g_1})$*-sec,* $(t_{g_2}, n+k, \epsilon_{g_2})$*-sec, then the constructions* x*-iMD*^{f_x,g_1,g_2}*,* y*-iMD*^{f_y,g_2}*,* c*-iMD*^{f_c,g_2} *with access to* $\{f_x, g_1, g_2\}, \{f_y, g_2\}, \{f_c, g_2\}$*, are* (t_x, L, ϵ_x)*-sec,* (t_y, L, ϵ_y)*-sec,* (t_c, L, ϵ_c)*-sec, such that:*

- *for* x*-iMD:* $\epsilon_x \leq \ell \cdot (\epsilon_{g_1} + \epsilon_{f_x}) \cdot \epsilon_{g_2}(2-\ell)$, *and* $t_x \leq (t_{f_x} + t_{g_1}) - 2(\ell \cdot \tau_{f_x} \tau_{g_1} + \tau_{g_2})$
- *for* y*-iMD:* $\epsilon_y \leq \ell \cdot \epsilon_{f_y} \epsilon_{g_2}(2-\ell)$, *and* $t_y \leq (t_{f_y} + t_{g_2}) - 2(\ell \cdot \tau_{f_y} \tau_{g_2} + \tau_{g_2})$
- *for* c*-iMD:* $\epsilon_c \leq \ell \cdot (\epsilon_{g_2} + \epsilon_{f_c}) \cdot \epsilon_{g_2}(2-\ell)$, *and* $t_c \leq (t_{f_c} + t_{g_2}) - 2(\ell \cdot \tau_{f_c} \tau_{g_2} + \tau_{g_2})$

where $\tau_{f_x}, \tau_{f_y}, \tau_{f_c}$ *are the costs of calling the compression functions* f_x, f_y, f_c*, and* τ_{g_1}, τ_{g_2} *are the costs of calling the integration functions* g_1, g_2*, respectively,* $L = max\{|M|, |M'|\}$*,* M' *is a 2nd pre-image of* M*,* $\ell = \lceil (L + 1 + \beta)/m \rceil$*,* β *is the length of the encoding of* L*, and* m *is the length of a single message block.*

Proof. Let A_x be $(t_{A_x}, L, \epsilon_{A_x})$-sec adversary against x-iMD that upon receiving a message M and a key K, it outputs a 2nd pre-image M' with probability ϵ_x, such that $x\text{-iMD}(K, M) = x\text{-iMD}(K, M')$, and $M \neq M'$. Also, let B_x be $(t_{B_x}, m+n, \epsilon_{B_x})$-s-sec adversary against the compression function f_x of x-iMD and G_{g_1}, G_{g_2} be $(t_{G_{g_1}}, k+m, \epsilon_{G_{g_1}})$-sec, $(t_{G_{g_2}}, k+n, \epsilon_{G_{g_2}})$-sec adversaries against the integration functions g_1, g_2, respectively. Let A_y, A_c and B_y, B_c be defined similarly for y-iMD, c-iMD, and f_y, f_c. We define composite adversaries D_x, D_y, D_c representing the composition of the compression and the integration functions $(f_x \star g_1), (f_y \star g_2), (f_c \star g_2)$, respectively, where $\star$ is a composition operation and is construction dependant. Formally, let the adversaries D_x, D_y, D_c be $(t_{D_x}, m+n, \epsilon_{D_x})$-sec/s-sec, $(t_{D_y}, m+n, \epsilon_{D_y})$-sec/s-sec, $(t_{D_c}, m+n, \epsilon_{D_c})$-sec/s-sec, which are formally defined in figure 4-5.

General Approach. First, we describe the general Sec attack and use a slightly clearer (but informal) notation than the one used in the definitions of figure 4-5. We refer to the composite adversary attacking the composition of the compression and the integration functions as D (this represents D_x, D_y, D_c). We also refer to the adversary attacking the iMD constructions as A (which represents A_x, A_y, A_c). We will later decompose D and distinguish between the different A's. The attack proceeds as follows:

1. D generates a random message $M \xleftarrow{\$} \{0,1\}^\lambda$ of length λ and pads it appropriately using Merkle-Damgård strengthening $M_{strengthened} = Pad_s(M)$, where Pad_s is as defined in figure 3-1.
2. D is then given an m-bit message block challenge $\hat{x} \xleftarrow{\$} \{0,1,\}^m$ which he needs to embed in $M_{strengthened}$.

3. D now needs to construct a key and a new message with $\hat{x}$ embedded in.

4. D chooses a random index $i \in \{1, 2, \ldots, \ell\}$, where $\ell = M_{strengthened}/m$ (the number of blocks in $M_{strengthened}$) and m is the length of a single block in $M_{strengthened}$.

5. D embeds $\hat{x}$ in $M_{strengthened}$ by replacing the i-th message block in $M_{strengthened}$ with $\hat{x}$, thus, for example, the modified message becomes $\hat{M}_{strengthened} = \{x_1, \ldots, x_{i-1}, \hat{x}, x_{i+1}, \ldots, x_\ell\}$, for $\ell \geq 5$ and $1 < i < \ell$.

6. D then generates a random key $K \xleftarrow{\$} \{0,1\}^k$, unpads $M_{strengthened}$ and submits both K and $M_{modified} = Pad_s(\hat{M}_{strengthened})^{-1}$ to A.

7. Since A is a 2nd pre-image adversary, it will return a 2nd pre-image with probability ϵ_A, that is $M' \xleftarrow{\$} A(K, M_{modified})$. This means that necessarily at least one of the blocks of M' collides with one of the blocks in $M_{modified}$. Then with probability $1/\ell$, the colliding block in $M_{modified}$ will be the i-th (embedded earlier) block, that is $M_{modified}[i] = \hat{x}$. Thus,

$$\mathbf{Adv}_D^{\text{sec/s-sec}} = 1/\ell \cdot \mathbf{Adv}_A^{\text{sec}}$$

where D's advantage is taken over the Sec and s-Sec games since it is a composition of B, whose advantage is taken over a s-Sec game, and G, whose advantage is taken over a Sec game. Solving for A, we then have:

$$\mathbf{Adv}_A^{\text{sec}} = \ell \cdot \mathbf{Adv}_D^{\text{sec/s-sec}}$$

Recall that A's advantage should also account for the final randomisation call of g_2 (this is absent in D's advantage since D is not affected by this call). However, this finalisation call of g_2 is made only once after processing all the ℓ message blocks, so we should not multiple g_2's Sec advantage by ℓ because it is used only in the last block to finalise the hash output. Thus, the advantage of A is:

$$\begin{aligned}
\mathbf{Adv}_A^{\text{sec}} &= \ell \cdot \mathbf{Adv}_D^{\text{sec/s-sec}} \cdot \left(\mathbf{Adv}_{G_{g_2}}^{\text{sec}} - (\ell - 1)\mathbf{Adv}_{G_{g_2}}^{\text{sec}}\right) \\
&= \ell \cdot \mathbf{Adv}_D^{\text{sec/s-sec}} \cdot \left(\mathbf{Adv}_{G_{g_2}}^{\text{sec}} - \left(\ell \cdot \mathbf{Adv}_{G_{g_2}}^{\text{sec}} - \mathbf{Adv}_{G_{g_2}}^{\text{sec}}\right)\right) \\
&= \ell \cdot \mathbf{Adv}_D^{\text{sec/s-sec}} \cdot \left(\mathbf{Adv}_{G_{g_2}}^{\text{sec}} - \ell \cdot \mathbf{Adv}_{G_{g_2}}^{\text{sec}} + \mathbf{Adv}_{G_{g_2}}^{\text{sec}}\right) \\
&= \ell \cdot \mathbf{Adv}_D^{\text{sec/s-sec}} \cdot \left(2 \cdot \mathbf{Adv}_{G_{g_2}}^{\text{sec}} - \ell \cdot \mathbf{Adv}_{G_{g_2}}^{\text{sec}}\right) \\
&= \ell \cdot \mathbf{Adv}_D^{\text{sec/s-sec}} \cdot \mathbf{Adv}_{G_{g_2}}^{\text{sec}}(2 - \ell)
\end{aligned}$$

To simplify the expression, here we multiple all the advantages by ℓ, and then subtract $\ell - 1$ from the advantage of the last g_2 call to remove the extra instances that were multiplied implicitly in the expression. We now calculate the $\mathbf{Adv}_D^{\text{sec/s-sec}}$ for every iMD construction and obtain the full Sec Advantage.

Before we derive the exact advantages of each iMD construction, it remains to discuss why we used the s-Sec notion on the compression functions f_x, f_y, f_c. Recall that we are deriving the advantages of adversary A against the iMD constructions in terms of the advantage of adversaries B and G against the compression functions f_x, f_y, f_c and the integration functions g_1, g_2, respectively. However, in the Sec attack above, while we were embedding $\hat{x}$ in the original message to produce a 2nd pre-image, there was no way to manipulate the corresponding chaining variable at the location where $\hat{x}$ was being embedded such that when it is concatenated with $\hat{x}$, the compression function will return a 2nd pre-image. The s-Sec notion (as defined in definition 4.4.2) solves this problem by fixing the chaining variable.

Advantage of x-iMD. Recall that in x-iMD the message block is pre-processed by g_1 before being passed to f_x. Thus, when we embed $\hat{x}$ in the i-th position of M by replacing m_i with $\hat{x}$, we get a 2nd pre-image if:

1. m_1 and $\hat{x}$ collide at g_1, that is $g_1(K, m_i) = g_1(K, \hat{x})$, or
2. m_1 and $\hat{x}$ do not collide at g_1, i.e., $g_1(K, m_i) \neq g_1(K, \hat{x})$, but then their g_1 outputs $g_1(K, m_i), g_1(K, \hat{x})$ collide at f_x, that is $f_x(g_1(K, m_i), X) = f_x(g_1(K, \hat{x}), X)$, while $g_1(K, m_i) \neq g_1(K, \hat{x})$, where $X \in \{0,1\}^n$ is some fixed string.

In case (1), we only need to account for the Sec advantage of g_1 (that g_1 will return a 2nd pre-image), while in case (2) we need to account for the s-Sec advantage of f_x. Thus the Sec advantage of A_x is:

$$\begin{aligned}
\mathbf{Adv}_{A_x}^{\text{sec}} &= \ell \cdot \mathbf{Adv}_{D_x}^{\text{sec/s-sec}} \cdot \mathbf{Adv}_{G_{g_2}}^{\text{sec}}(2-\ell) \\
&= \ell \cdot \left(\mathbf{Adv}_{G_{g_1}}^{\text{sec}} + \mathbf{Adv}_{B_x}^{\text{s-sec}}\right) \cdot \mathbf{Adv}_{G_{g_2}}^{\text{sec}}(2-\ell) \\
&\leq \ell \cdot (\epsilon_{G_{g_1}} + \epsilon_{B_x}) \cdot \epsilon_{G_{g_2}}(2-\ell) \\
&\leq \ell \cdot (\epsilon_{g_1} + \epsilon_{f_x}) \cdot \epsilon_{g_2}(2-\ell)
\end{aligned}$$

Advantage of y-iMD. Similar analysis can be used to derive the Sec advantage of adversary A_y in the Sec game against y-iMD, except here g_1 is not involved which makes the calculation even easier. Also, in y-iMD the integration function g_2 is applied to the chaining variable input of f_y and so does not affect the s-Sec advantage of f_y,

meaning that the advantage of g_2 does not have to be accounted for in the overall advantage of the game. That is,

$$\begin{aligned}\mathbf{Adv}^{\text{sec}}_{A_y} &= \ell \cdot \mathbf{Adv}^{\text{s-sec}}_{D_y} \cdot \mathbf{Adv}^{\text{sec}}_{G_{g_2}}(2-\ell) \\ &= \ell \cdot \mathbf{Adv}^{\text{s-sec}}_{B_y} \cdot \mathbf{Adv}^{\text{sec}}_{G_{g_2}}(2-\ell) \\ &\leq \ell \cdot \epsilon_{B_y}\epsilon_{G_{g_2}}(2-\ell) \\ &\leq \ell \cdot \epsilon_{f_y}\epsilon_{g_2}(2-\ell)\end{aligned}$$

Advantage of c-iMD. The advantage of c-iMD resembles that of x-iMD except g_1 in x-iMD is replaced by g_2 in c-iMD. That is, the block x_i is first embedded in the message, but a second pre-image is found if:

1. m_i and $\hat{x}$ collide at f_c, or
2. m_i and $\hat{x}$ do not collide at f_c, such that $f_c(m_i, X) \neq f_c(\hat{x}, X)$, where $m_i \neq \hat{x}$, but their f_c outputs collide at g_2, that is $g_2(K, f_c(m_i, X)) = g_2(K, f_c(\hat{x}, X))$, where $X \in \{0,1\}^n$ is some fixed string.

Case (1) implies that the s-Sec game of f_c is successful, and case (2) implies that the Sec game of g_2 is successful. Succinctly,

$$\begin{aligned}\mathbf{Adv}^{\text{sec}}_{A_c} &= \ell \cdot \mathbf{Adv}^{\text{sec/s-sec}}_{D_c} \cdot \mathbf{Adv}^{\text{sec}}_{G_{g_2}}(2-\ell) \\ &= \ell \cdot \left(\mathbf{Adv}^{\text{sec}}_{G_{g_2}} + \mathbf{Adv}^{\text{s-sec}}_{B_c}\right) \cdot \mathbf{Adv}^{\text{sec}}_{G_{g_2}}(2-\ell) \\ &\leq \ell \cdot (\epsilon_{G_{g_2}} + \epsilon_{B_c}) \cdot \epsilon_{G_{g_2}}(2-\ell) \\ &\leq \ell \cdot (\epsilon_{g_2} + \epsilon_{f_c}) \cdot \epsilon_{g_2}(2-\ell)\end{aligned}$$

The time complexity (the adversarial running time) of D_x, D_y, D_c is the execution time of A_x, A_y, A_c in addition to two evaluations of x-iMD or y-iMD or c-iMD (to hash the message and its 2nd pre-image), where the latter implicitly include a finalisation call to g_2. That is, $t_{D_x} = t_{A_x} + 2(\ell \cdot \tau_{g_1}\tau_{f_x} + \tau_{g_2})$. Since $t_{D_x} = t_{B_x} + t_{G_{g_1}}$, and solving for t_{A_x}, we get $t_{A_x} = t_{B_x} + t_{G_{g_1}} - 2(\ell \cdot \tau_{g_1}\tau_{f_x} + \tau_{g_2})$, where $\tau_{f_x}, \tau_{g_1}, \tau_{g_2}$ are the costs of calling f_x, g_1, g_2, respectively. t_{A_y} and t_{A_c} are calculated similarly. □

4.4.3 Pre-image Resistance (Pre)

Unlike CR and Sec, it turned out that providing Pre proofs in the standard model is inherently difficult. Thus, in this section, we instead prove that the iMD constructions are Pre in the random oracle model (where f_x, f_y, f_c, g_1, g_2 are modelled as $\mathcal{RO}s$), but first we define two notions that we will use in the proof.

Adversary $D_x(\hat{x})$
$K \xleftarrow{\$} \mathcal{K}; M \xleftarrow{\$} \{0,1\}^\lambda$
$x_1, x_2, \ldots, x_\ell \leftarrow Pad_s(M)$
$y_0 = y'_0 = IV$
for $j = 1$ **to** $j = \ell$ **do**
$z_j = f_x(g_{1,K}(x_j), y_{j-1})$
$y_i = z_j$
end for; $i \xleftarrow{\$} \{1, 2, \ldots, \ell\}$
$Pad_s(M) \leftarrow \text{replace}(x_i, \hat{x})$
$\tilde{M} \leftarrow \text{unpad}(Pad_s(M))$
$M' \leftarrow A(K, \tilde{M})$
for $j = 1$ **to** $j = \ell$ **do**
$z'_j = f_x(g_{1,K}(x'_j), y'_{j-1})$
$y'_j = z'_j$
if $x_i \neq x'_j \wedge z_i = z'_j$
then return (x'_i, y'_{i-1})
else return $\perp$
end for

Adversary $D_y(\hat{x})$
$K \xleftarrow{\$} \mathcal{K}; M \xleftarrow{\$} \{0,1\}^\lambda$
$x_1, x_2, \ldots, x_\ell \leftarrow Pad_s(M)$
$y_0 = y'_0 = IV$
for $j = 1$ **to** $j = \ell$ **do**
$z_j = f_y(x_j, g_{2,K}(y_{j-1}))$
$y_j = z_j$
end for; $i \xleftarrow{\$} \{1, 2, \ldots, \ell\}$
$Pad_s(M) \leftarrow \text{replace}(x_i, \hat{x})$
$\tilde{M} \leftarrow \text{unpad}(Pad_s(M))$
$M' \leftarrow A(K, \tilde{M})$
for $j = 1$ **to** $j = \ell$ **do**
$z'_j = f_y(x'_j, g_{2,K}(y'_{j-1}))$
$y'_j = z'_j$
if $x_i \neq x'_j \wedge z_i = z'_j$
then return (x'_i, y'_{i-1})
else return $\perp$
end for

Adversary $D_c(\hat{x})$
$K \xleftarrow{\$} \mathcal{K}; M \xleftarrow{\$} \{0,1\}^\lambda$
$x_1, x_2, \ldots, x_\ell \leftarrow Pad_s(M)$
$y_0 = y'_0 = IV$
for $j = 1$ **to** $j = \ell$ **do**
$z_j = g_{2,K}(f_c(x_j, y_{j-1}))$
$y_j = z_j$
end for; $i \xleftarrow{\$} \{1, 2, \ldots, \ell\}$
$Pad_s(M) \leftarrow \text{replace}(x_i, \hat{x})$
$\tilde{M} \leftarrow \text{unpad}(Pad_s(M))$
$M' \leftarrow A(K, \tilde{M})$
for $j = 1$ **to** $j = \ell$ **do**
$z'_j = g_{2,K}(f_c(x'_j, y'_{j-1}))$
$y'_j = z'_j$
if $x_i \neq x'_j \wedge z_i = z'_j$
then return (x'_i, y'_{i-1})
else return $\perp$
end for

Figure 4-5: (Composite) Adversaries D_x, D_y, D_c in the Sec game

Definition 4.4.4 (Full inversion). *The full invasion of a function F, denoted $inv(F)$, implies retrieving an input of F from knowledge of a corresponding output. Formally, let F be a function, then given Y, $inv(F(Y)) = M$, where $F(M) = Y$.*

Definition 4.4.5 (Partial inversion). *The partial invasion of a function F, denoted p-$inv(F)$, implies retrieving part of the function's input from knowledge of only its output. Formally, let $F(M) = Y$ be a function returning Y on input M, and let $|M| > \lambda$, then p-$inv_\lambda(F(Y)) = M^\lambda$, where M^λ is the λ most significant bits of M.*

A particular output Y may have more than one image $M_1, M_2, \ldots, M_n$, such that $F(M_1) = F(M_2) = \cdots = F(M_n) = Y$, thus the full and partial inversions of Y may return any (part of any) image. Full and partial inversions are not particularly new notions, but we formally define them here since we will explicitly use them in the proof.

Theorem 4.4.6 (Pre of x-iMD, y-iMD, c-iMD). *Let the keyless compression functions $f_x : f_y : f_c : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^n$ and the keyed integration functions $g_1 : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^m, g_2 : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be modelled as Random Oracles, then x-iMD^{f_x,g_1,g_2}, y-iMD^{f_y,g_2}, c-iMD^{f_c,g_2} with access to $\{f_x, g_1, g_2\}, \{f_y, g_2\}, \{f_c, g_2\}$, respectively, are (t_x, q_x, ϵ_x)-pre, (t_y, q_y, ϵ_y)-pre, (t_c, q_c, ϵ_c)-pre, such that:*

- $\epsilon_x = q_x^5(\ell-1)/2^{2(2m+n)}$, *and* $t_x = q_{A_x}\left((\ell-1)\tau_{f_x}\tau_{g_1} + \tau_{g_2}\right) + \tau_{f_x}\tau_{g_1}$
- $\epsilon_y = q_y^5(\ell-1)/2^{2(m+n)+n}$, *and* $t_y = q_{A_y}\tau_{g_2}\left((\ell-1)\tau_{f_y} + 1\right) + \tau_{f_y}$
- $\epsilon_c = q_c^5(\ell-1)/2^{2(m+2n)}$, *and* $t_c = q_{A_c}\tau_{g_2}\left((\ell-1)\tau_{f_c} + 1\right) + \tau_{f_c}\tau_{g_2}$

for any number of queries q_x, q_y, q_c, where ℓ is the number of blocks in the message to be inverted, which is known, $\tau_{f_x}, \tau_{f_y}, \tau_{f_c}$ are the costs of calling the compression functions f_x, f_y, f_c and τ_{g_1}, τ_{g_2} are the costs of calling the integration functions g_1, g_2.

Proof. Let A_x, A_y, A_c be $(t_{A_x}, q_{A_x}, \epsilon_{A_x})$-pre, $(t_{A_y}, q_{A_y}, \epsilon_{A_y})$-pre, $(t_{A_c}, q_{A_c}, \epsilon_{A_c})$-pre adversaries against x-iMD, y-iMD, c-iMD, respectively. Given a hash value $x\text{-iMD}_K(M) = Y$ and a random key $K \in \{0,1\}^k$, A_x inverts Y after q_{A_x} queries to x-iMD with probability ϵ_{A_x}; adversaries A_y and A_c function similarly. In order for adversaries A_x, A_y, A_c to be able to invert a given Y, they have to invert the underlying primitives from which x-iMD, y-iMD, c-iMD are built. That is, x-iMD iteratively calls a composite function consisting of the keyless compression function $f_x : \{0,1\}^{m+n} \to \{0,1\}^n$ and the keyed integration function $g_1 : \{0,1\}^{k+m} \to \{0,1\}^m$, then finalises the hashing process with a call to the integration function $g_2 : \{0,1\}^{k+n} \to \{0,1\}^n$. Similarly, y-iMD accesses $f_y : \{0,1\}^{m+n} \to \{0,1\}^n$ and $g_2 : \{0,1\}^{k+n} \to \{0,1\}^n$ and iteratively processes the message by calling the composition of f_y and g_2 then finalises the process with a single call to g_2; c-iMD functions similarly. We assume that A_x, A_y, A_c know the length of M. This also means that A_x, A_y, A_c know how many blocks there will be after padding, let ℓ be this number. In all constructions, A_x, A_y, A_c should first be able to invert the finalisation call of g_2, then invert the composition of the compression and integration functions for as many blocks as the message to be recovered M consists of. Finally, A_x, A_y, A_c need to only partially invert the first application of the compression/integration composite function because they already know part of it, which is the IV, and are only interested in recovering the message part. Thus, the inversion probabilities are calculated as follows[6]:

$$\begin{aligned}
\mathbf{Adv}^{\text{pre}}_{A_x} &= \Pr[inv(g_2)] \cdot (\ell-1)\Pr[inv(f_x) \cdot inv(g_1)] \cdot \Pr[p\text{-}inv_m(f_x) \cdot inv(g_1)] \\
&= \frac{q_{A_x}}{2^n} \cdot (\ell-1)\left(\frac{q_{A_x}}{2^{m+n}} \cdot \frac{q_{A_x}}{2^m}\right) \cdot \left(\frac{q_{A_x}}{2^m} \cdot \frac{q_{A_x}}{2^m}\right) \\
&= \frac{q_{A_x}}{2^n} \cdot (\ell-1)\frac{q_{A_x}^2}{2^{2m+n}} \cdot \frac{q_{A_x}^2}{2^{2m}} = \frac{q_{A_x}}{2^n} \cdot \frac{q_{A_x}^4(\ell-1)}{2^{4m+n}} = \frac{q_{A_x}^5(\ell-1)}{2^{2(2m+n)}}
\end{aligned}$$

[6]Since the $\mathcal{RO}$ does not exhibit collisions, there is only 1 input that will produce a given output, so the probability that an adversary A will find the input for a giving output at output space (range) of size 2^n is $1/2^n$, and after making q_A queries to the $\mathcal{RO}$, this probability becomes $q_A/2^2$.

$$\begin{aligned}\mathbf{Adv}^{\text{pre}}_{A_y} &= \Pr[inv(g_2)] \cdot (\ell-1)\Pr[inv(f_y)\cdot inv(g_2)] \cdot \Pr[p\text{-}inv_m(f_y)] \\ &= \frac{q_{A_y}}{2^n} \cdot (\ell-1)\left(\frac{q_{A_y}}{2^{m+n}} \cdot \frac{q_{A_y}}{2^n}\right) \cdot \frac{q_{A_y}}{2^m} = \frac{q^5_{A_y}(\ell-1)}{2^{2(m+n)+n}}\end{aligned}$$

$$\begin{aligned}\mathbf{Adv}^{\text{pre}}_{A_c} &= \Pr[inv(g_2)] \cdot (\ell-1)\Pr[inv(f_c)\cdot inv(g_2)] \cdot \Pr[p\text{-}inv_m(f_c)\cdot inv(g_2)] \\ &= \frac{q_{A_c}}{2^n} \cdot (\ell-1)\left(\frac{q_{A_c}}{2^{m+n}} \cdot \frac{q_{A_c}}{2^n}\right) \cdot \left(\frac{q_{A_c}}{2^m} \cdot \frac{q_{A_c}}{2^n}\right) = \frac{q^5_{A_c}(\ell-1)}{2^{2(m+2n)}}\end{aligned}$$

We multiply all the probabilities because this is a join probability, i.e., all the inversions (events) have to occur in order for the Pre attack to success. Let $\tau_{f_x}, \tau_{f_y}, \tau_{f_c}$ be the costs of calling f_x, f_y, f_c and τ_{g_1}, τ_{g_2} be the costs of calling g_1, g_2. Then the running times of adversaries A_x, A_y, A_c are:

$$\begin{aligned}t_{A_x} &= q_{A_x}\tau_{g_2} + q_{A_x}(\ell-1)\tau_{f_x}\tau_{g_1} + \tau_{f_x}\tau_{g_1} = q_{A_x}\left((\ell-1)\tau_{f_x}\tau_{g_1} + \tau_{g_2}\right) + \tau_{f_x}\tau_{g_1} \\ t_{A_y} &= q_{A_y}\tau_{g_2} + q_{A_y}(\ell-1)\tau_{f_y}\tau_{g_2} + \tau_{f_y} = q_{A_y}\tau_{g_2}\left((\ell-1)\tau_{f_y} + 1\right) + \tau_{f_y} \\ t_{A_c} &= q_{A_c}\tau_{g_2} + q_{A_c}(\ell-1)\tau_{f_c}\tau_{g_2} + \tau_{f_c}\tau_{g_c} = q_{A_c}\tau_{g_2}\left((\ell-1)\tau_{f_c} + 1\right) + \tau_{f_c}\tau_{g_2}\end{aligned}$$

The running times reflect the fact that both f_x and g_1, in case of x-iMD, and f_c and g_2 in the case of c-iMD, are called to invert the first message block in M since both the compression and the integration functions in x-iMD and c-iMD need to be inverted in order to obtain the message block input of the first iteration. However, in y-iMD, g_2 is placed at the chaining variable input of f_y, thus it only suffices to partially invert f_y to obtain the first message block. Although the adversaries actually only partially invert this first iteration, they still make the call to the compression/integration functions, which, obviously, should be accounted for in the running time of the adversaries. □

4.5 Summary

As a result to numerous recent attacks against the conventional keyless hash functions, designers are now more inclined to construct their hash functions in the keyed setting, which creates families of hash functions and, generally, provides better security guarantees. However, while it would be uneconomical to abandon today's popular keyless hash functions, converting them to keyed functions without modifying their keyless primitives seems difficult. In this chapter, we proposed an easy and simple fix to this problem by introducing a separate function accompanying the compression function to handle the key input independently outside the compression function, we call this setting the *integrated-key* setting. We further proposed 3 integrated-key constructions and proved that they are collision, pre-image and 2nd pre-image resistant.

Chapter 5

Indifferentiability of the iMD Constructions

In chapter 4 we introduced and formalised the integrated-key setting, where keyless hash functions are seamlessly transformed into keyed variants, which creates keyed hash functions out of "unmodified" keyless primitives. In this chapter, we adopt the iMD constructions, proposed in chapter 4, and prove that they are indifferentiable from random oracle. We show in details how to develop such indifferentiability proofs in the integrated-key setting. Our indifferentiability proof is generic and can be applicable to other hash functions constructed in this setting if they exhibit sufficient structural similarities to the iMD constructions.

5.1 Introduction

A classical problem in the hash function literature is how to formally argue about the security of a hash function without the presence of keys [134]. This problem, along with numerous cryptanalytic results [154, 155, 156] on some of the most popular keyless hash functions, clearly demonstrate that there are inherent weaknesses in the keyless design approach. A simple solution is to instead shift to *keyed* hash functions, but adopting new hash functions is neither economical nor straightforward. In most cases, it is much more convenient (and cheaper) to patch an existing hash function rather than shifting to a new one, but then the underlying building blocks of a keyless hash function will need to undertake non-trivial modifications to adapt for the key input. That is, a typical keyless hash function $H^h : \{0,1\}^* \rightarrow \{0,1\}^n$, with access to a keyless compression function $f : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$, accepts a message $M \in \{0,1\}^*$ of variable-

length, divides it into m-bit blocks (pads if necessary), and hash the blocks iteratively by repeatedly calling f. In contrast, the compression function in a typical keyed setting is keyed $h : \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$, where the key $K \in \{0,1\}^k$ is processed alongside the other inputs. Clearly, converting f to h, without major modifications to the underlying structure of f, is difficult, if at all possible. In chapter 4 we proposed the integrated-key approach as a moderate solution to this problem which seamlessly turns keyless hash functions into keyed ones without *touching* the underlying keyless compression function. In this setting, a dedicated mixing function is introduced to handle the key input independently outside the compression function. One of the most important properties that hash functions should preserve is being indifferentiable from a random oracle ($\mathcal{RO}$). In this chapter, we show that the iMD constructions preserve this property.

Chapter Outline. This chapter is organised as follows. In section 5.2 we recall the iMD constructions from chapter 4, then in section 5.3 we briefly describe the indifferentiability framework due to Coron *et al.* [48], but for a comprehensive discussion about this framework see section 2.3. Section 5.4 is the main part of this chapter where we provide a detailed indifferentiability proof of the iMD constructions.

5.2 The iMD Constructions (Recalled)

As shown in chapter 3, most of the popular hash functions (e.g., MD5, SHA-1) are based on the (keyless) Merkle-Damgård construction [111, 55]. Thus, in chapter 4, we proposed several integrated-key constructions, called the iMD constructions, based on the Merkle-Damgård construction and proved that they preserve several important security properties. In this chapter, we further show that the iMD constructions are indifferentiable from $\mathcal{RO}$. Although both the compression[1] and the integration functions can be visualised as a single entity, to obtain more accurate indifferentiability bounds, we make an explicit separation between the two and show how indifferentiability proofs are generally carried out in the integrated-key setting. Figure 5-1 recalls the definitions of the x-iMD, y-iMD, c-iMD constructions from chapter 4 with slight modification in x-iMD, where $Pad_s(.)$ is a suffix-free padding (e.g., as defined in figure 3-1).

Formally, $x\text{-iMD}^{f_x,g_x}, y\text{-iMD}^{f_y,g_y}, c\text{-iMD}^{f_c,g_c} : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$, where $f_x, f_y, f_c : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n, g_x : \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m, g_y, g_c : \{0,1\}^k \times$

[1] Most compression functions utilise the Davies-Meyer (DM) or the Matyas-Meyer-Oseas (MMO) constructions, but we will assume that this is implicit and is hidden inside the (keyless) compression function, so the output of the compression function $f : x \times y \rightarrow c$ is actually $c = c' \oplus x$ for DM and $c = c' \oplus y$ for MMO, where c' is the output of the compression function before applying DM or MMO.

x-iMD$^{f_x,g_x}(K,M)$:	y-iMD$^{f_y,g_y}(K,M)$:	c-iMD$^{f_y,g_c}(K,M)$:
$M_1...M_\ell \leftarrow Pad_s(M)$	$M_1...M_\ell \leftarrow Pad_s(M)$	$M_1...M_\ell \leftarrow Pad_s(M)$
$y_0 = IV$	$y_0 = IV$	$y_0 = IV$
for $i = 1$ **to** ℓ **do**	**for** $i = 1$ **to** ℓ **do**	**for** $i = 1$ **to** ℓ **do**
$y_i = f_x(g_x(K, M_i), y_{i-1})$	$y_i = f_y(M_i, g_y(K, y_{i-1}))$	$y_i = g_c(K, f_y(M_i, y_{i-1}))$
return $y = g_x(K, 0^{m-n}\|\|y_\ell)$	**return** $y = g_y(K, y_\ell)$	**return** $y = g_c(K, y_\ell)$

Figure 5-1: Pseudocode for the x-iMD, y-iMD, c-iMD constructions

$\{0,1\}^n \rightarrow \{0,1\}^n$. Instead of giving x-iMD a dedicated function to handle the last finalising call, we use its own integration function g_x and pad its input with $m-n$ 0 bits. This should not affect its security arguments, but will simplify the indifferentiability proof (where we ignore the time it takes to do this padding). We do not have this issue with y-iMD and c-iMD, so the definitions of y-iMD and c-iMD in figure 5-1 are identical to those in figure 4-1 of chapter 4.

5.3 The Indifferentiability Framework

Proofs in the standard model (where adversarial resources are limited) can become extremely difficult for sufficiently complex cryptosystem. This fact motivated Bellare and Rogaway [26] to propose a formalism of the well known Random Oracle Model (ROM) which assumes the presence of a publicly accessible ideal primitive that when given an input, returns a random output. However, recent separation results [40, 119, 20, 41, 108, 100] questioned the soundness of this model since, in practice, $\mathcal{RO}$s are being instantiated by hash functions, which may not always behave like $\mathcal{RO}$s. Consequently, based on the notion of indifferentiability by Maurer *et al.* [108], in [48] Coron *et al.* introduced their hash function indifferentiability framework where a hash function is proven indifferentiable from $\mathcal{RO}$ and thus is expected to behave like one. In this framework, a distinguisher D (which plays the role of an adversary) is given oracle access to two (separate) systems, we call the first system the *real system*, consisting of a compression function and the construction under consideration (which needs to be proven indifferentiable from $\mathcal{RO}$), and we call the second system the *ideal system*, consisting of a $\mathcal{RO}$ and a simulator (where the latter simulates the behaviour of the compression function of the real system at the ideal system). Indifferentiability proofs can be conducted in either the Random Oracle Model (ROM) or the Ideal Cipher Model (ICM). The difference between these two approaches is that in the ROM, the ideal compression function at the real system is modelled as $\mathcal{RO}$, while it is modelled as an ideal block-cipher in the ICM. In the latter case, the ideal block-cipher can receive both forward and inverse queries because block-ciphers are invertible). Regardless of

the adopted model, proofs in the indifferentiability framework proceed in two steps (as discussed in section 2.3): first, we propose the simulator (simulating a compression function or an ideal cipher), and then we prove that D's view is similar when it interacts with the real system as it is when it interacts with the ideal system (D cannot distinguish between the two systems); see [43] for examples and discussions of how indifferentiability proofs are developed in both ROM and ICM. The formal definition of the indifferentiability framework is as follows [48] (where the constructions under consideration is referred to as a Turing machine C, the ideal compression function as $\mathcal{H}$ and the random oracle as $\mathcal{F}$):

Definition 5.3.1 (Indifferentiability from $\mathcal{RO}$). *A Turing machine C with oracle access to an ideal primitive $\mathcal{H}$ is said to be (t_D, t_S, q, ϵ)-indifferentiable from an ideal primitive $\mathcal{F}$ if there exists a simulator S such that for any distinguisher D it holds that:*

$$|\Pr[D^{C,\mathcal{H}} = 1] - \Pr[D^{\mathcal{F},S} = 1]| < \epsilon$$

The simulator S has oracle access to $\mathcal{F}$ and runs in time at most t_S. The distinguisher runs in time at most t_D and makes at most q queries to $C, \mathcal{H}, \mathcal{F}$ or S. C is said to be (computationally) indifferentiable from $\mathcal{F}$ if ϵ is a negligible function of the security parameter.

5.4 Indifferentiability of the iMD Constructions

In this section, we prove that the x-iMD^{f_x,g_x}, y-iMD^{f_y,g_y}, c-iMD^{f_c,g_c} constructions are indifferentiable from $\mathcal{RO}$ in the Ideal Cipher Model (ICM), when the compression functions f_x, f_y, f_c and the integration functions g_x, g_y, g_c are modelled as ideal block-ciphers $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c, \mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$, respectively. In [50], it was shown that the ROM and the ICM are equivalent; that is, given a scheme secure in the ROM, it remains secure in the ICM when the $\mathcal{RO}$s is replaced by ideal ciphers, and conversely, given a scheme secure in the ICM, it remains secure in the ROM when the ideal ciphers are replaced by $\mathcal{RO}$s. We adopt the ICM because most of the popular hash functions are (explicitly or implicitly) based on block-ciphers.

The constructions x-iMD$^{\mathcal{H}_x,\mathcal{G}_x}$, x-iMD$^{\mathcal{H}_y,\mathcal{G}_y}$, x-iMD$^{\mathcal{H}_c,\mathcal{G}_c}$ treat $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ and $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$ as black-boxes such that upon receiving a message M, they partition it into equally sized blocks $x_1, \ldots, x_\ell$ and process each block in turn through $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ and $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$. Formally, in c-iMD, $\mathcal{H}_c$ receives a message block $x_i \in \{0,1\}^m$ and a chaining variable (or IV) $y_i \in \{0,1\}^n$, and returns a temporary variable $c_i \in \{0,1\}^n$, which is immediately given to $\mathcal{G}_c$ along with the key $K \in \{0,1\}^k$ to finally return $z_i \in \{0,1\}^n$ (note

that for a sequence of consecutive blocks, $y_{i+1} = z_i$). In x-iMD, on the other hand, the integration function $\mathcal{G}_x$ is called before $\mathcal{H}_x$ to process K and x_i which will return c_i that, along with y_i, will be given to $\mathcal{H}_x$ to return z_i. y-iMD is similar to x-iMD, except in this case $\mathcal{G}_y$ processes K with y_i instead of x_i, where the latter is given to $\mathcal{H}_y$ along with c_i (the output of $\mathcal{G}_y$) to produce z_i.

In the ideal system, we introduce two simulators, the compression function and the integration function simulators $S^{\mathcal{F}}, R^{\mathcal{F}}$, both with oracle access to the random oracle $\mathcal{F}$. Figure 5-2 depicts the indifferentiability games for the x-iMD, y-iMD, c-iMD constructions, showing how the distinguisher D accesses each system (where x-iMD, y-iMD, c-iMD are denoted by by $\hat{C}_x, \hat{C}_y, \hat{C}_c$, respectively).

5.4.1 The Distinguisher

The distinguisher D is an adversary with oracle access to two systems (the real and ideal systems) and whose aim is to prove that these systems can be distinguished from each other (but not necessarily finding out which system is which, rather all D aims for is to show that the two systems behave differently). Precisely, D is given a *blind* oracle access to the systems, meaning that there is an imaginary *barrier* between D and the systems preventing D from seeing the systems, as depicted in figure 5-2. Trying to fool the systems, D carries out a set of tests on one system by repeatedly (and strategically) querying that system's different components and observes the responses, D sets the success conditions and outputs 1 if these tests succeed, 0 otherwise[2]. Simultaneously, D carries out the same set of tests on the other system, observes the responses and similarly outputs 1 or 0. D fails if both systems behaved consistently and then the two systems are said to be indifferentiable from each other. D may send either forward or inverse queries but format them differently depending on which component they are sent to; thus, D indeed knows what type of components it is interacting with (e.g., compression function, integration function etc.) but it does not know to which system they belong. In some cases, D can choose to exclusively interact with a particular system for a period of time, but it will not be able to choose which system that would be. Forward queries may be sent to any component in any system, but inverse queries may only be sent to $\mathcal{H}_i, \mathcal{G}_i, S_i^{\mathcal{F}}, R_i^{\mathcal{F}}, i \in \{x, y, c\}$; these components can receive inverse queries because we model the compression and the integration functions as ideal block-ciphers where these are clearly invertible. When D sends an inverse query to a particular component, it interacts with an inverse variant of that component A^{-1}, where $A \in \{\mathcal{H}_i, \mathcal{G}_i, S_i^{\mathcal{F}}, R_i^{\mathcal{F}}\}$. D communicates with the systems through three query

[2]In other words, D outputs 1 if it thinks that it is interacting with, e.g., the idea system, otherwise it outputs 0 (or vice versa depending on D's definition).

channels, Ch_1, Ch_2, Ch_3, which are connected to three interfaces, if_1, if_2, if_3, at the barrier separating D from the real and ideal systems.

- Ch_1: D uses this channel to interact with the constructions $\hat{C}_i^{\mathcal{H}_i,\mathcal{G}_i}, i \in \{x, y, c\}$ and the random oracle $\mathcal{F}$. This channel supports only forward queries of the form (K, M), where $K \in \{0,1\}^k$ and $M \in \{\{0,1\}^m\}^*$, and delivers back to D the response $z \in \{0,1\}^n$. To simplify the proof, we assume that $|M|$ is a multiple of m, we also ignore padding rules, but the proof still holds with padding included.

- Ch_2: D uses this channel to interact with the ideal compression functions $\mathcal{H}_x, \mathcal{H}_y$, $\mathcal{H}_c$ and their simulators $S_x^{\mathcal{F}}, S_y^{\mathcal{F}}, S_c^{\mathcal{F}}$. It supports both forward and inverse queries of the form $(\rightarrow, c, y)$ and $(\leftarrow, z, c)$ for $S_x^{\mathcal{F}}$, $(\rightarrow, x, c)$ and $(\leftarrow, z, x)$ for $S_y^{\mathcal{F}}$, and $(\rightarrow, x, y)$ and $(\leftarrow, c, x)$ for $S_c^{\mathcal{F}}$, then delivers back to D the appropriate responses, namely, it returns z for queries $(\rightarrow, c, y)$, $(\rightarrow, x, c)$, y for queries $(\leftarrow, z, c)$, $(\leftarrow, c, x)$, and c for query $(\leftarrow, z, x)$, $(\rightarrow, x, y)$.

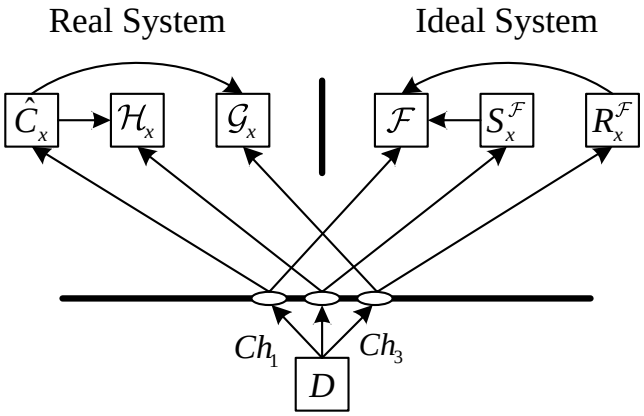
- Ch_3: D uses this channel to communicate with the ideal integration functions $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$ and their simulators $R_x^{\mathcal{F}}, R_y^{\mathcal{F}}, R_c^{\mathcal{F}}$. Like Ch_2, this channel also allows both forward and inverse queries of the form $(\rightarrow, x, K)$ and $(\leftarrow, c, K)$ for $R_x^{\mathcal{F}}$, $(\rightarrow, y, K)$ and $(\leftarrow, c, K)$ for $R_y^{\mathcal{F}}$, and $(\rightarrow, c, K)$ and $(\leftarrow, z, K)$ for $R_c^{\mathcal{F}}$, then delivers back to D the appropriate responses: x for query $(\leftarrow, c, K)$, y for query $(\leftarrow, c, K)$, c for queries $(\rightarrow, x, K)$,$(\rightarrow, y, K)$,$(\leftarrow, z, K)$, and z for query $(\rightarrow, c, K)$.

Each channel is split into two channels past its corresponding interface, one split for the real system and another for the ideal system. Hence, when an interface if_x receives a query from D, it creates two identical copies of that query and sends them off the other end through the channel's two splits, unless D chooses to exclusively interacts with a single system for a period of time, in which case the interface chooses that system at random (D cannot choose which system to interact with) and starts sending D's queries to the components of that system until D advises otherwise. That is, the interfaces cannot randomly switch between the systems without an explicit request from D.

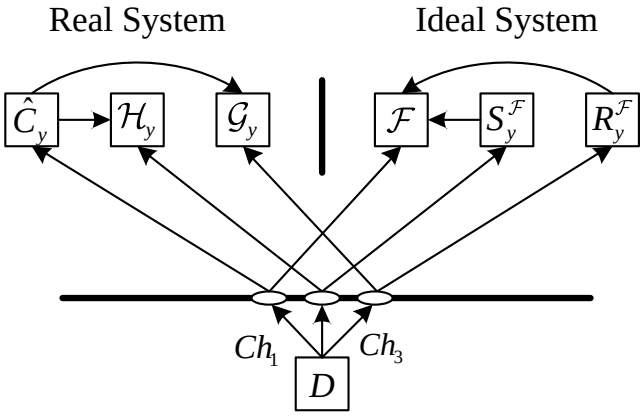
5.4.2 The Proof

In this section, we adopt the Ideal Cipher Model (ICM) and prove that the constructions x-iMD, y-iMD, c-iMD, with access to the ideal ciphers $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c, \mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$ are indifferentiable from a $\mathcal{RO}$, except with negligible probability.

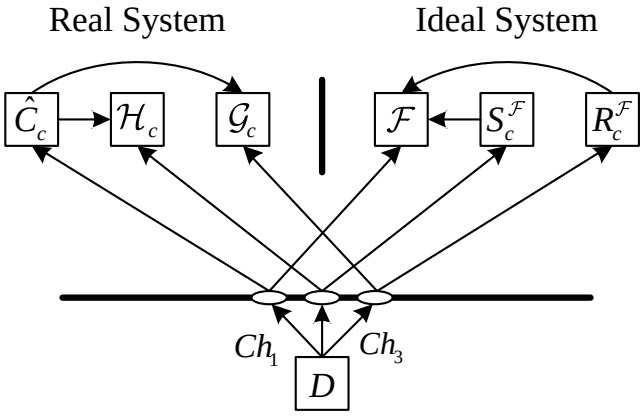
Theorem 5.4.1 (Indifferentiability of x-iMD, y-iMD, c-iMD). *The block-cipher based Integrated-key constructions x-iMD$^{\mathcal{H}_x,\mathcal{G}_x}$, y-iMD$^{\mathcal{H}_y,\mathcal{G}_y}$, c-iMD$^{\mathcal{H}_c,\mathcal{G}_c}$, with oracle*



(a) D's view in the x-iMD game



(b) D's view in the y-iMD game



(c) D's view in the c-iMD game

Figure 5-2: The interaction between D and the real/ideal systems in the x-iMD, y-iMD, c-iMD games

access to the ideal block-ciphers $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$*, and the ideal integration functions* $\mathcal{G}_x : \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m, \mathcal{G}_y, \mathcal{G}_c : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$*, modelled as ideal block-ciphers, are* $(t_D, t_S, q_1, q_2, q_3, \epsilon_x)$*-indifferentiable* $(t_D, t_S, q_1, q_2, q_3, \epsilon_y)$*-indifferentiable* $(t_D, t_S, q_1, q_2, q_3, \epsilon_c)$*-indifferentiable from a random oracle* $\mathcal{F}$ *in the ideal cipher model for* $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ *and* $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$*, any* t_D*,* $t_S \leq (q_1 \cdot L/m + q_2 + q_3) \cdot (m+n)$*, any number of queries* q_1, q_2, q_3 *sent by* D *to* x*-iMD*$^{\mathcal{H}_x,\mathcal{G}_x}$*,* y*-iMD*$^{\mathcal{H}_y,\mathcal{G}_y}$*,* c*-iMD*$^{\mathcal{H}_c,\mathcal{G}_c}$*, and:*

$$
\begin{aligned}
\epsilon_x &\leq \left(2^n(5((q_1 \cdot L/m) + q_2 + q_3) + 6((q_1 \cdot L/m) + q_2 + q_3)^2)\right)/2^{2n} \\
&+ \left((q_1 \cdot L/m) + q_2 + q_3\right)/2^{2n} + \left(((q_1 \cdot L/m) + q_2 + q_3)\right)/2^m \\
\epsilon_y &\leq \left(7((q_1 \cdot L/m) + q_2 + q_3) + 6((q_1 \cdot L/m) + q_2 + q_3)^2\right)/2^n \\
&+ \left(2^2((q_1 \cdot L/m) + q_2 + q_3)^2 + ((q_1 \cdot L/m) + q_2 + q_3)\right)/2^{m+n} \\
\epsilon_c &\leq \left((q_1 \cdot L/m + q_2 + q_3)(3 \cdot 2^m + 2^{m+1}(q_1 \cdot L/m + q_2 + q_3) + 1))\right)/2^{m+n} \\
&+ \left(2^n(q_1 \cdot L/m + q_2 + q_3)^2\right)/2^{m+n} \\
&+ \left(2(q_1 \cdot L/m + q_2 + q_3) + 3(q_1 \cdot L/m + q_2 + q_3)^2\right)/2^n
\end{aligned}
$$

where L *is the maximum length of the query* q_1*.*

Proof. We prove the indifferentiability by means of a hybrid argument. We adopt the game-playing approach [28, 48] (described in section 2.3.1) and prove that consecutive games are indifferentiable from each other, stating the distinguishing probability when applicable. Each game represents a state of the system which then evolves as the proof progresses through the games. We start with $G(1)$, Game 1, which represents the ideal system (consisting of the $\mathcal{RO}$ and simulators of the ideal compression and integration functions) and finish with $G(8)$ (consisting of the construction and the ideal compression and integration functions), Game 8, which represents the real system. We denote x-iMD$^{\mathcal{H}_x,\mathcal{G}_x}$, y-iMD$^{\mathcal{H}_y,\mathcal{G}_y}$, c-iMD$^{\mathcal{H}_c,\mathcal{G}_c}$ by $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$, respectively, and prove that they are indifferentiable from the random oracle $\mathcal{F}$. Below we integrate the indifferentiability proofs of the three constructions noting that the real/ideal systems of each proof may consist of slightly different components, which we will often state explicitly.

The Simulators. We first propose the required simulators. The proof requires a total of six simulators proposed in the ideal systems to simulate components in the real systems. Simulators $S_x^{\mathcal{F}}, S_y^{\mathcal{F}}, S_c^{\mathcal{F}}$ simulate the ideal compression functions $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ and simulators $R_x^{\mathcal{F}}, R_y^{\mathcal{F}}, R_c^{\mathcal{F}}$ simulate the ideal integration functions $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$, respectively; all simulators have oracle access to $\mathcal{F}$. The proof for $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}$ uses $S_x^{\mathcal{F}}, R_x^{\mathcal{F}}$,

the proof for $\hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}$ uses $S_y^{\mathcal{F}}, R_y^{\mathcal{F}}$, and the proof for $\hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$ uses $S_c^{\mathcal{F}}, R_c^{\mathcal{F}}$. Figure 4 graphically illustrates the input/output notation for each construction, we will use this notation extensively throughout the proof. Each simulator pair $S_i^{\mathcal{F}}$-$R_i^{\mathcal{F}}$ cooperatively

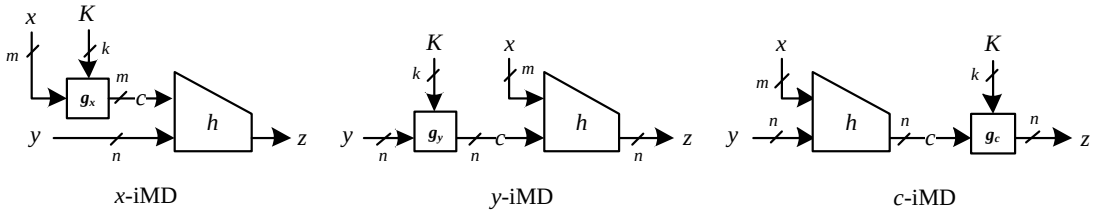


Figure 5-3: Input/output notation of the x-iMD, y-iMD, c-iMD constructions

maintains a table $\mathcal{T}_i^K$, that is initially empty $\perp$ but gradually grow as D interacts with $S_i^{\mathcal{F}}$-$R_i^{\mathcal{F}}$, where $i \in \{x, y, c\}$. Since integrated-key hash functions are actually families of hash functions, where the individual function members are indexed by different keys $K \in \mathcal{K}$, $S_i^{\mathcal{F}}$ and $R_i^{\mathcal{F}}$ will maintain different tables $\mathcal{T}_i^K$ for different keys (members). Without loss of generality, here we will assume that we are interacting with a single hash function member and that the key K is fixed throughout the proof. As illustrated in figure 5-4, all tables contain 5-tuple records of the form $(x_i^{s_l,p}, y_i^{s_l,p}, c_i^{s_l,p}, z_i^{s_l,p}, t_i^{s_l,p})$, where $x_i \in \{0,1\}^m, y_i \in \{0,1\}^n$ are the message block and chaining variable (or IV), $z_i \in \{0,1\}^n$ is the output of the ideal ciphers $\mathcal{H}_x, \mathcal{H}_y$ (in the case of x-iMD and y-iMD) or the output of the idea integration function $\mathcal{G}_c$ (in the case of c-iMD), and c_i is the output of the ideal integration functions $\mathcal{G}_x, \mathcal{G}_y$ (in the case of x-iMD and y-iMD) or the output of the ideal cipher $\mathcal{H}_c$ (in the case of c-iMD), such that:

$$c_i \in \begin{cases} \{0,1\}^m & \text{if } c_i \in \mathcal{T}_x^K \\ \{0,1\}^n & \text{if } c_i \in \mathcal{T}_y^K \cup \mathcal{T}_c^K \end{cases}$$

The index $i \in \{0, 1, \dots\}$ of a record specifies the location of the tuple in tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, the tag $t_i \in \{\perp, 0, 1\}$ indicates whether the tuple is part of a sequence, and if it does s_l indicates to which sequence (indexed by $l \in \{\perp, 0, 1, \dots\}$) this tuple belongs, and $p \in \{\perp, 0, 1, \dots\}$ specifies the exact location of the tuple in the sequence s_l. A sequence is an ordered list of tuples such that $z_a^{s_l,p-1} = y_b^{s_l,p}$, and is rooted by the j-th tuple where $y_j^{s_l,0} = IV$; note that if two consecutive tuples in a sequence do not have to be consecutive in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ (if $z_a^{s_l,p-1} = y_b^{s_l,p}$, then "$a$" does not have to be "$b-1$", while they are still indexed in succession in the sequence s_l). A tuple belonging to a sequence is called *sequenced* tuple, otherwise it is *singular*. To keep track of the number of sequences, $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ maintain counters $\tilde{C}_x^K, \tilde{C}_y^K, \tilde{C}_c^K$ which are initially

set to 0 but are incremented every time a tuple with $y_{\_} = IV$ is encountered (recall that we use the notation _ (underscore) to denote an arbitrary tuple in a table). As per the definitions of $S_i^{\mathcal{F}}, R_i^{\mathcal{F}}, i \in \{x, y, c\}$ below, a sequence has to be rooted by a tuple where $y_{\_}^{s_l,0} = IV$, otherwise a sequence may not be formed, even if there are tuples in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ that can be connected. Clearly, $t = \perp$ implies $l = p = \perp$; in fact, t here acts as a switch to activate or deactivate the index s_l (however, we may sometimes drop the s_l, p index when referencing tuples if it is not needed). Furthermore, tuples in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ are required to be complete (there are no tuples with missing fields). Figures 5-5 and 5-6 formally define simulators $S_i^{\mathcal{F}}, R_i^{\mathcal{F}}, i \in \{x, y, c\}$ and their inverse variants; these simulators are based on two simple rule:

1. The c value is always generated by a $\mathcal{RO}$. If it was given in a query, c is further processed through a $\mathcal{RO}$, that is $\mathcal{F}(c)$.

2. Unless given in a query, the z value is always generated by a $\mathcal{RO}$.

Below we describe the simulators in more details, recall that all simulators receive both forward and inverse queries.

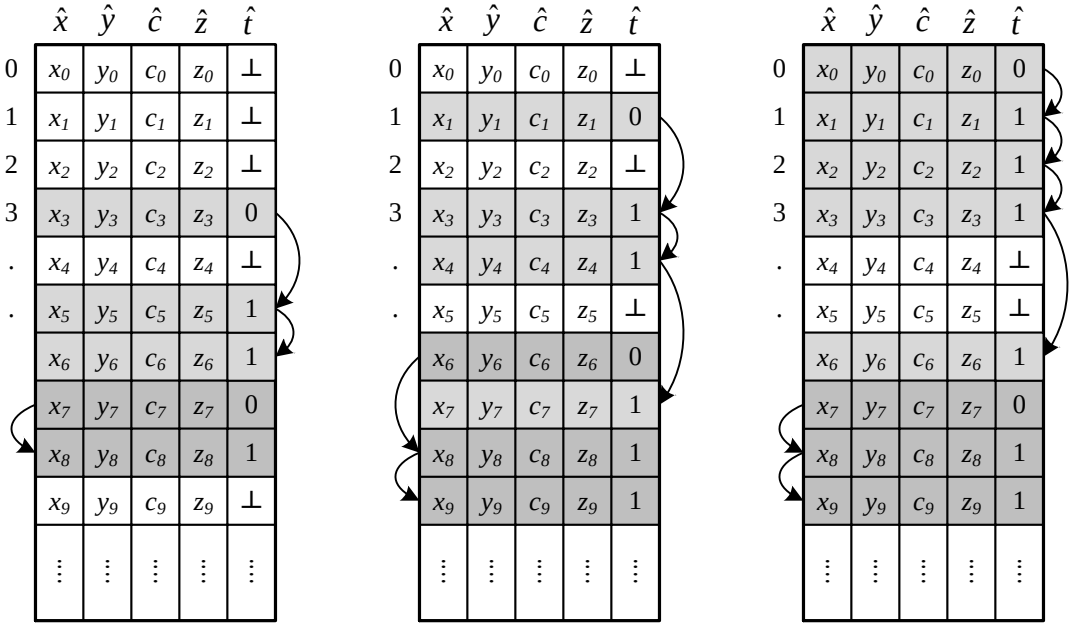


Figure 5-4: Samples of tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ illustrating how tuples and sequences are organised and linked

Simulators $S_x^{\mathcal{F}}$ and $(S_x^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, c, y)$, $S_x^{\mathcal{F}}$ searches $\mathcal{T}_x^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $c_i = c$ and $y_i = y$, if found, it returns z_i. Otherwise, $S_x^{\mathcal{F}}$ generates a new value for x uniformly at random $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$, then $S_x^{\mathcal{F}}$ proceeds to generate z as follows:

   1(a). If $y = IV$, then $S_x^{\mathcal{F}}$ makes the query $\mathcal{F}(\mathcal{F}(c), y)$ to the random oracle $\mathcal{F}$ to obtain z, and sets $t = 0$, $l = \tilde{C}_K + 1, p = 0$.

   1(b). If there is a tuple $(x_i^{s_l,p}, y_i^{s_l,p}, c_i^{s_l,p}, z_i^{s_l,p}, t_i^{s_l,p})$ in $\mathcal{T}_x^K$ such that $y = z_i^{s_l,p}$, then $S_x^{\mathcal{F}}$ obtains z by querying the random oracle $\mathcal{F}(\mathcal{F}(c), y)$, while setting $t = 1$ and indexes the new tuple by $s_l, p + 1$.

   1(c). Otherwise, $S_x^{\mathcal{F}}$ obtains z by querying $\mathcal{F}(\mathcal{F}(c), y)$, and sets $t = \perp$, with s_l deactivated.

2. On inverse query $(\leftarrow, z, c)$, $(S_x^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_x^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $z_i = z$ and $c_i = c$, if found, it returns y_i. Otherwise, $(S_x^{\mathcal{F}})^{-1}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$, where $\mathcal{T}_x^K(\hat{y})$ and $\mathcal{T}_x^K(\hat{z})$ extract all the y and z values in the table $\mathcal{T}_x^K$ which, along with IV, will be excluded from the value assigned to y, and similarly generates x uniformly at random $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$, while setting $t = \perp$, with s_l deactivated.

Simulators $R_x^{\mathcal{F}}$ and $(R_x^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, K, x)$, $R_x^{\mathcal{F}}$ searches $\mathcal{T}_x^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $x_i = x$, if found, it returns c_i. Otherwise, $R_x^{\mathcal{F}}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$, then query $\mathcal{F}$ on x to obtain c, that is $c \leftarrow \mathcal{F}(K, x)$, and finally query $\mathcal{F}$ on y and c to obtain z, that is $z \leftarrow \mathcal{F}(y, c)$, while setting $t = \perp$ with s_l deactivated.

2. On inverse query $(\leftarrow, K, c)$, $(R_x^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_x^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $c_i = c$, if found, it returns x_i. Otherwise, $(R_x^{\mathcal{F}})^{-1}$ generates both x and y uniformly at random: $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$, $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$, and then generates z by querying $\mathcal{F}$ that is $z \leftarrow \mathcal{F}(\mathcal{F}(c), y)$. Finally, it sets $t = \perp$ with the index s_l deactivated.

Simulators $S_y^{\mathcal{F}}$ and $(S_y^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, x, c)$, $S_y^{\mathcal{F}}$ searches $\mathcal{T}_y^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $x_i = x$ and $c_i = c$, if found, it returns z_i. Otherwise, $S_y^{\mathcal{F}}$ generates y uniformly

at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$. Then $S_y^{\mathcal{F}}$ obtains z by querying $\mathcal{F}$ as follows: $z \leftarrow \mathcal{F}(x, \mathcal{F}(c))$, while setting $t = \perp$, with s_l deactivated.

2. On inverse query $(\leftarrow, z, x)$, $(S_y^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_y^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $z_i = z$ and $x_i = x$, if found, it returns c_i. Otherwise, $(S_y^{\mathcal{F}})^{-1}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$, and queries $\mathcal{F}$ to obtain c such that $c \leftarrow \mathcal{F}(K, y)$, while setting $t = \perp$, with s_l deactivated.

Simulators $R_y^{\mathcal{F}}$ and $(R_y^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, K, y)$, $R_y^{\mathcal{F}}$ searches $\mathcal{T}_y^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $y_i = y$, if found, it returns c_i. Otherwise, $R_y^{\mathcal{F}}$ generates x uniformly at random $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$, and query $\mathcal{F}$ to generate $c \leftarrow \mathcal{F}(K, y)$, then $R_y^{\mathcal{F}}$ proceeds to generate z as follows (recall that z has to be generated to complete the new tuple):

   1(a). If $y = IV$, then $R_y^{\mathcal{F}}$ makes the query $\mathcal{F}(x, c)$ to the random oracle $\mathcal{F}$ to obtain z, and sets $t = 0$, $l = \tilde{C}_K + 1$ (increment the counter $\tilde{C}$) and $p = 0$.

   1(b). If there is a tuple $(x_i^{s_l,p}, y_i^{s_l,p}, c_i^{s_l,p}, z_i^{s_l,p}, t_i^{s_l,p})$ in $\mathcal{T}_c^K$ such that $y = z_i^{s_l,p}$, then $R_y^{\mathcal{F}}$ obtains z by querying the random oracle $\mathcal{F}(T||x, c)$, where:

$$T = x_{\_}^{s_l,0} || x_{\_}^{s_l,1} || \ldots || x_{\_}^{s_l,p-1} || x_{\_}^{s_l,p}$$

   which is a chain of queries rooted by the tuple indexed by $s_l, 0$. Once the new tuple is created, $R_y^{\mathcal{F}}$ indexes it by $s_l, p+1$ while setting $t^{s_l,p+1} = 1$.

   1(c). Otherwise, $R_y^{\mathcal{F}}$ obtains z by querying $\mathcal{F}(x, c)$, and sets $t = \perp$, with s_l deactivated.

2. On inverse query $(\leftarrow, K, c)$, $(R_y^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_y^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $c_i = c$, if found, it returns y_i. Otherwise, $(R_y^{\mathcal{F}})^{-1}$ generates both x and y uniformly at random: $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_y^K(\hat{x})\}$, $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$, and then generates z by querying $\mathcal{F}$ such that $z \leftarrow \mathcal{F}(x, \mathcal{F}(c))$. Finally, it sets $t = \perp$ with the index s_l deactivated.

Simulators $S_c^{\mathcal{F}}$ and $(S_c^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, x, y)$, $S_c^{\mathcal{F}}$ searches $\mathcal{T}_c^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $x_i = x$ and $y_i = y$, if found, it returns c_i. Otherwise, $S_c^{\mathcal{F}}$ creates a new tuple by querying $\mathcal{F}$ for c on input (x, y), that is $c \leftarrow \mathcal{F}(x, y)$, then $S_c^{\mathcal{F}}$ proceeds to generate z as follows (recall that z has to be generated to complete the new tuple):

1(a). If $y = IV$, then $S_c^{\mathcal{F}}$ makes the query $\mathcal{F}(K, c)$ to the random oracle $\mathcal{F}$ to obtain z, and sets $t = 0$, $l = \tilde{C}_K + 1$ (increment the counter $\tilde{C}$) and $p = 0$.

1(b). If there is a tuple $(x_i^{s_l,p}, y_i^{s_l,p}, c_i^{s_l,p}, z_i^{s_l,p}, t_i^{s_l,p})$ in $\mathcal{T}_c^K$ such that $y = z_i^{s_l,p}$, then $S_c^{\mathcal{F}}$ obtains z by querying the random oracle $\mathcal{F}(K, T||x)$, where:

$$T = x_{\_}^{s_l,0}||x_{\_}^{s_l,1}||\ldots||x_{\_}^{s_l,p-1}||x_{\_}^{s_l,p}$$

which is a chain of queries rooted by the tuple indexed by $s_l, 0$. Once the new tuple is created, $S_c^{\mathcal{F}}$ indexes it by $s_l, p+1$ while setting $t^{s_l,p+1} = 1$.

1(c). Otherwise, $S_c^{\mathcal{F}}$ obtains z by querying $\mathcal{F}(K, c)$, and sets $t = \perp$, with s_l deactivated.

2. On inverse query $(\leftarrow, c, x)$, $(S_c^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_c^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $x_i = x$ and $c_i = c$, if found, it returns y_i. Otherwise, $(S_c^{\mathcal{F}})^{-1}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$, where $\mathcal{T}_c^K(\hat{y})$ and $\mathcal{T}_c^K(\hat{z})$ extract all the y and z values in the table $\mathcal{T}_c^K$ which, along with IV, will be excluded from the value assigned to y, obtains z by querying $\mathcal{F}(K, \mathcal{F}(c))$, and sets $t = \perp$, with s_l deactivated.

Simulators $R_c^{\mathcal{F}}$ and $(R_c^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, K, c)$, $R_c^{\mathcal{F}}$ searches $\mathcal{T}_c^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $c_i = c$, if found, it returns z_i. Otherwise, $R_c^{\mathcal{F}}$ generates both x and y uniformly at random: $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_c^K(\hat{x})\}$, $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$, and then queries $\mathcal{F}$ to generate $z \leftarrow \mathcal{F}(K, \mathcal{F}(c))$. Finally, it sets $t = \perp$ with the index s_l deactivated.

2. On inverse query $(\leftarrow, K, z)$, $(R_c^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_c^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $z_i = z$, if found, it returns c_i. Otherwise, $(R_c^{\mathcal{F}})^{-1}$ generates x and y uniformly at random: $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_c^K(\hat{x})\}$, $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$, and then generates c by querying $\mathcal{F}$ on input (x, y): $c \leftarrow \mathcal{F}(x, y)$. Finally, it sets $t = \perp$ with the index s_l deactivated.

The Indifferentiability Proof. We now construct the games. Throughout the proof, P_i^x, P_i^y, P_i^c denote the probabilities that D outputs 1 in game $G(i)$ of the indifferentiability proofs of $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$, respectively. Figure 5-7 depicts the state of the systems in each game.

Simulator $S_x^{\mathcal{F}} : (\rightarrow, c, y)$
if $(c_i, y_i) \in \mathcal{T}_x^K \wedge (c, y) = (c_i, y_i)$ **then**
  return z_i
else $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$
  if $y = \text{IV}$ **then**
    $t = 0,\ l = \tilde{C}_k + 1,\ p = 0$
    return $z \leftarrow \mathcal{F}(\mathcal{F}(c), y)$
  else if $y = z_i^{s_l,p}, z_i^{s_l,p} \in \mathcal{T}_x^K$ **then**
    $t = 1, p = p + 1$
    return $z \leftarrow \mathcal{F}(\mathcal{F}(c), y)$
  else $z \leftarrow \mathcal{F}(\mathcal{F}(c), y),\ t = \perp$

Simulator $(S_x^{\mathcal{F}})^{-1} : (\leftarrow, z, c)$
if $(z_i, c_i) \in \mathcal{T}_x^K \wedge (z, c) = (z_i, c_i)$
  return y_i
else return
  $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$
  $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\},\ t = \perp$

Simulator $S_y^{\mathcal{F}} : (\rightarrow, x, c)$
if $(x_i, c_i) \in \mathcal{T}_y^K \wedge (x, c) = (x_i, c_i),$ **then**
  return z_i
else
  $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$
  return $z \leftarrow \mathcal{F}(x, \mathcal{F}(c))$
  $t = 0$

Simulator $(S_y^{\mathcal{F}})^{-1} : (\leftarrow, z, x)$
if $(z_i, x_i) \in \mathcal{T}_y^K \wedge (z, x) = (z_i, x_i),$ **then**
  return c_i
else
  $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$
  return $c \leftarrow \mathcal{F}(K, y)$
  $t = \perp$

Simulator $S_c^{\mathcal{F}} : (\rightarrow, x, y)$
if $(x_i, y_i) \in \mathcal{T}_c^K \wedge (x, y) = (x_i, y_i),$ **return** c_i
else return $c \leftarrow \mathcal{F}(x, y)$
  if $y = \text{IV},$ **then**
    $t = 0,\ l = \tilde{C}_k + 1,\ p = 0,\ z \leftarrow \mathcal{F}(K, c)$
  else if $y = z_i^{s_l,p}, z_i^{s_l,p} \in \mathcal{T}_c^K,$ **then**
    $t = 1, p = p + 1$
    $T = x^{s_l,0}\_||x^{s_l,1}\_|| \ldots ||x^{s_l,p-1}\_||x^{s_l,p}\_$
    $z \leftarrow \mathcal{F}(K, T||x)$
  else $z \leftarrow \mathcal{F}(K, c),\ t = \perp$

Simulator $(S_c^{\mathcal{F}})^{-1} : (\leftarrow c, x)$
if $(c_i, x_i) \in \mathcal{T}_c^K \wedge (c, x) = (c_i, x_i),$ **return** y_i
else return
  $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$
  $z \leftarrow \mathcal{F}(K, \mathcal{F}(c)),\ t = \perp$

Figure 5-5: Simulators $S_x^{\mathcal{F}}, (S_x^{\mathcal{F}})^{-1}, S_y^{\mathcal{F}}, (S_y^{\mathcal{F}})^{-1}, S_c^{\mathcal{F}}, (S_c^{\mathcal{F}})^{-1}$

Simulator $R_x^{\mathcal{F}} : (\rightarrow, K, x)$
if $x_i \in \mathcal{T}_x^K \wedge x = x_i$, **return** c_i
else
$t = \perp$
$c \leftarrow \mathcal{F}(K, x)$
$y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$
return $z \leftarrow \mathcal{F}(y, c)$

Simulator $(R_x^{\mathcal{F}})^{-1} : (\leftarrow, K, c)$
if $c_i \in \mathcal{T}_x^K \wedge c = c_i$, **return** x_i
else
$t = \perp$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$
$y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$
return $z \leftarrow \mathcal{F}(\mathcal{F}(c), y)$

Simulator $R_y^{\mathcal{F}} : (\rightarrow, K, y)$
if $y_i \in \mathcal{T}_y^K \wedge y = y_i$, **return** c_i
else return $c \leftarrow \mathcal{F}(K, y)$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_y^K(\hat{x})\}$
if $y = IV$ **then**
$t = 0,\ l = \tilde{C}_k + 1,\ p = 0,\ z \leftarrow \mathcal{F}(x, c)$
else if $y = z_i^{s_l,p}, z_i^{s_l,p} \in \mathcal{T}_y^K$ **then**
$t = 1, p = p + 1$
$T = x_{\_}^{s_l,0} || x_{\_}^{s_l,1} || \ldots || x_{\_}^{s_l,p-1} || x_{\_}^{s_l,p}$
$z \leftarrow \mathcal{F}(T||x, c)$
else $z \leftarrow \mathcal{F}(x, c),\ t = \perp$

Simulator $(R_y^{\mathcal{F}})^{-1} : (\leftarrow, K, c)$
if $c_i \in \mathcal{T}_y^K \wedge c = c_i$, **return** y_i
else return
$y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_y^K(\hat{x})\}$
$z \leftarrow \mathcal{F}(x, \mathcal{F}(c)),\ t = \perp$

Simulator $R_c^{\mathcal{F}} : (\rightarrow, K, c)$
if $c_i \in \mathcal{T}_c^K \wedge c = c_i$, **return** z_i
else $t = \perp,\ x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_c^K(\hat{x})\})$
$y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$
return $z \leftarrow \mathcal{F}(K, \mathcal{F}(c))$

Simulator $(R_c^{\mathcal{F}})^{-1} : (\leftarrow, K, z)$
if $z_i \in \mathcal{T}_c^K \wedge z = z_i$, **return** c_i
else $t = \perp,\ x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_c^K(\hat{x})\}$
$y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$
return $c \leftarrow \mathcal{F}(x, y)$

Figure 5-6: Simulators $R_x^{\mathcal{F}}, (R_x^{\mathcal{F}})^{-1}, R_y^{\mathcal{F}}, (R_y^{\mathcal{F}})^{-1}, R_c^{\mathcal{F}}, (R_c^{\mathcal{F}})^{-1}$

Game 1. This is the $\mathcal{RO}$ game where D is exclusively interacting with the ideal system. Let

$$\begin{aligned} P_1^x &= \Pr[D^{\mathcal{F}, S_x^{\mathcal{F}}, R_x^{\mathcal{F}}} = 1] \\ P_1^y &= \Pr[D^{\mathcal{F}, S_y^{\mathcal{F}}, R_y^{\mathcal{F}}} = 1] \\ P_1^c &= \Pr[D^{\mathcal{F}, S_c^{\mathcal{F}}, R_c^{\mathcal{F}}} = 1] \end{aligned}$$

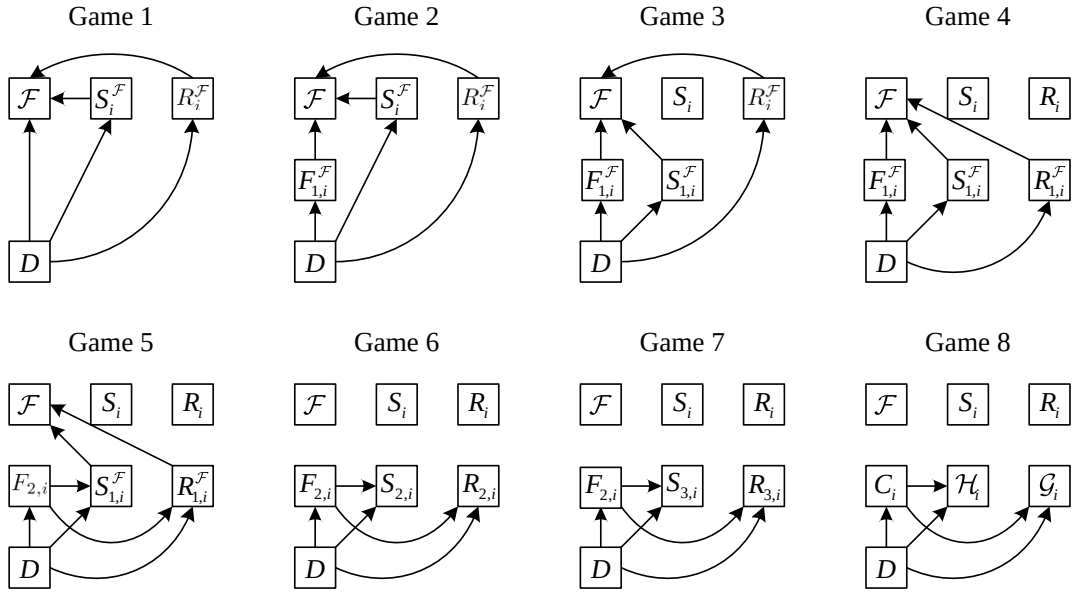


Figure 5-7: A depiction showing how the system's state evolves through the games in the indifferentiability proof of $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$

Game 2. In this game we introduce dummy relay algorithms $F_{1,x}, F_{1,y}, F_{1,c}$ placed between D and $\mathcal{F}$ in the $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$ proofs, respectively. These algorithms relay queries from D to $\mathcal{F}$ and relay responses back from $\mathcal{F}$ to D. Obviously, the view of D in $G(2)$ is not affected by the introduction of $F_{1,x}, F_{1,y}, F_{1,c}$, thus:

$$\begin{aligned} P_2^x &= \Pr[D^{F_{1,x}^{\mathcal{F}},S_x^{\mathcal{F}},R_x^{\mathcal{F}}} = 1] = P_1^x \\ P_2^y &= \Pr[D^{F_{1,y}^{\mathcal{F}},S_y^{\mathcal{F}},R_y^{\mathcal{F}}} = 1] = P_1^y \\ P_2^c &= \Pr[D^{F_{1,c}^{\mathcal{F}},S_c^{\mathcal{F}},R_c^{\mathcal{F}}} = 1] = P_1^c \end{aligned}$$

Game 3. In this game we introduce slightly modified replicas of $S_x^{\mathcal{F}}, S_y^{\mathcal{F}}, S_c^{\mathcal{F}}$ (and their inverse variants), still with oracle access to $\mathcal{F}$. D will now interact with the modified simulators $S_{1,x}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$ which, upon receiving a query, create a new tuple (x, y, c, z, t) but explicitly abort if any of the following failure conditions is satisfied:

1. On forward queries $S_{1,x}^{\mathcal{F}}(\rightarrow, c, y), S_{1,y}^{\mathcal{F}}(\rightarrow, x, c), S_{1,c}^{\mathcal{F}}(\rightarrow, x, y)$, the simulators create the new tuple (x, y, c, z, t), but the following collisions occur:

   1(a). Fixed point: either in $S_{1,x}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$, it is the case that $z = IV$, or in $S_{1,x}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$, it is the case that $z = y$.

   1(b). Prefix collision: in $S_{1,x}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}$, it is the case that $z = y_i$ for some $y_i \in \mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{y})$.

1(c). Internal collision: there is a tuple $(x_i, y_i, c_i, z_i, t_i)$ in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ such that:

i. when $S_{1,x}^{\mathcal{F}}$ receives $(\rightarrow, c, y)$, the following hold: $z = z_i \wedge (c, y) \neq (c_i, y_i)$.

ii. when $S_{1,y}^{\mathcal{F}}$ receives $(\rightarrow, x, c)$, the following hold: $z = z_i \wedge (x, c) \neq (x_i, c_i i)$.

iii. when $S_{1,c}^{\mathcal{F}}$ receives $(\rightarrow, x, y)$, the following hold: $c = c_i \wedge (x, y) \neq (x_i, y_i)$, or $z = z_i \wedge c \neq c_i$.

2. On inverse queries $(S_{1,x}^{\mathcal{F}})^{-1}(\leftarrow, z, c), (S_{1,y}^{\mathcal{F}})^{-1}(\leftarrow, z, x), (S_{1,c}^{\mathcal{F}})^{-1}(\leftarrow, c, x)$, the simulators create the new tuple (x, y, c, z, t), but the following collisions occur:

2(a). Fixed point: in $(S_{1,c}^{\mathcal{F}})^{-1}$, it is the case that $z = IV$.

2(b). Prefix collision: in $(S_{1,x}^{\mathcal{F}})^{-1}, (S_{1,y}^{\mathcal{F}})^{-1}, (S_{1,c}^{\mathcal{F}})^{-1}$, it is the case that $z = y_i$ for some $y_i \in \mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{y})$.

2(c). Internal collision: there is a tuple $(x_i, y_i, c_i, z_i, t_i)$ in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ such that:

i. when $(S_{1,y}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, z, x)$, the following hold: $c = c_i \wedge y \neq y_i$.

ii. when $(S_{1,c}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, c, x)$, the following hold: $z = z_i \wedge c \neq c_i$.

2(d). Partial query collision: there is a tuple $(x_i, y_i, c_i, z_i, t_i)$ in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ such that:

i. when $(S_{1,x}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, z, c)$, the following hold: $c \neq c_i \wedge z = z_i$.

ii. when $(S_{1,y}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, z, x)$, the following hold: $x \neq x_i \wedge z = z_i$.

iii. when $(S_{1,c}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, c, x)$, the following hold: $x \neq x_i \wedge c = c_i$.

Clearly, failure conditions 1(a) and 2(a) are similar. In this case, there are two types of fixed points, (i) when $z = IV$, and (ii) when $z = y$. Generating $z_i = IV$ may indeed happen but with very low probability since in the case of $S_{1,x}^{\mathcal{F}}$ and $S_{1,y}^{\mathcal{F}}$, there is only one combination of c and y or x and c, respectively, that will cause $z = IV$; similarly, in the case of $S_{1,c}^{\mathcal{F}}$, there is one value of c (combined with the fixed key K) that will cause $z = IV$. That is, unless given as part of a query, z is always generated by a $\mathcal{RO}$ (according to the definitions of the simulators), but while generating it, we cannot instruct the $\mathcal{RO}$ to exclude IV from the possible values it may return for z. Similarly, the fixed point $z = y$ (the output collides with the input) can only happen when querying $S_{1,x}^{\mathcal{F}}$ and $S_{1,c}^{\mathcal{F}}$. In both cases, y is given as part of the query, but we do not know what $\mathcal{RO}$ query would generate the given y, and since z (in both cases) is generated by a $\mathcal{RO}$, it may be the case that the inputs used to generate z are the ones that would generate y. Even though in $S_{1,x}^{\mathcal{F}}$ the y value is actually part of the $\mathcal{R}$ input that generates z, there is nothing stopping the $\mathcal{RO}$ from output y (part of its input) as this is equally random. Clearly, none of the inverse simulators

$(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (S^{\mathcal{F}}_{1,c})^{-1}$ may output $z = y$ since z is given as part of their queries (and in the case of $(S^{\mathcal{F}}_{1,c})^{-1}$, y is generated excluding the value of z). When calculating the probability of all these failure conditions we should account for queries sent to both $S^{\mathcal{F}}_i$ and $R^{\mathcal{F}}_i$ since they cooperatively add tuples to $\mathcal{T}^K_i$ whenever they are queried, where $i \in \{x, y, c\}$. However, since it is possible that new queries will not create new tuples (if they match existing tuples), the probability is upper bounded by $q_2 + q_3/2^n$, where q_2 are queries sent to $S^{\mathcal{F}}_{1,i}, (S^{\mathcal{F}}_{1,i})^{-1}$ and q_3 are queries sent to $R^{\mathcal{F}}_{1,i}, (R^{\mathcal{F}}_{i,i})^{-1}$, where $i \in \{x, y, c\}$. We do not account for q_1 in this probability because the q_1 queries are sent to F_1, F_2, F_3 which, at this stage, do not contribute in updating the tables $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$, respectively, and so they will not affect the probability of finding collisions there.

Failure conditions 1(b) and 2(b) are similar. However, in this case z may collide with any y_i belonging to any tuple in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$, but since not all queries will add new tuples to the table, this probability is upper bounded by the birthday attack [128], that is $\leq (q_2 + q_3)^2/2^n$. When creating a new tuple (x, y, c, z, t) upon receiving a query, there are two types of prefix collisions, either $y = z_i$ or $z = y_j$ for some $z_i, y_j \in \mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$. In the first type, if z_i is sequenced (part of a sequence) and $y = z_i$, then the simulators connect the tuple of the newly queried y with the tuple of the matching z_i (the i-th tuple) and no collision occurs (recall that only simulators $S^{\mathcal{F}}_{1,x}, S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_{1,y}$ can create/extend sequences because they are the only simulators who receive y as part of their queries). But what if z_i is not sequenced? In this case, the simulators will treat it as a sequenced tuple that is part of a sequence consisting of only one tuple, itself (the i-th tuple). Connecting the newly created tuple with the i-th tuple will create an "unrooted" sequence, one that has no route tuple with $y = IV$, but this will not affect the indifferentiability proofs, as we will see later. This leaves the second type of prefix collision, $z = y_j$, which applies to $S^{\mathcal{F}}_{1,x}, S^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ and their inverse variants. To see why this is the case, we discuss how such collision may be generated in $\mathcal{T}^K_x$, which is maintained by $S^{\mathcal{F}}_{1,x}, (S^{\mathcal{F}}_{1,x})^{-1}, R^{\mathcal{F}}_{1,x}, (R^{\mathcal{F}}_{1,x})^{-1}$, similar argument apply for $\mathcal{T}^K_y$ and $\mathcal{T}^K_c$. Simulator $S^{\mathcal{F}}_{1,x}$ receives y as part of its query and then creates a new tuple if necessary, but other simulators have no way to exclude the y value that $S^{\mathcal{F}}_{1,x}$ received when they generate their z values as the latter is generated by a $\mathcal{RO}$ which may output the value of the previously queried y with probably $(q_2 + q_3)^2/2^n$.

Failure conditions 1(c) and 2(c), on the other hand, allude to a more fundamental problem when the simulators receive the c and z values from D rather then generating them. These values should ideally be generated by a $\mathcal{RO}$, but on queries to $S^{\mathcal{F}}_{1,x}, (S^{\mathcal{F}}_{1,x})^{-1}, S^{\mathcal{F}}_{1,y}, (S^{\mathcal{F}}_{1,c})^{-1}, (R^{\mathcal{F}}_{1,x})^{-1}, (R^{\mathcal{F}}_{1,y})^{-1}, R^{\mathcal{F}}_{1,c}$, the value of c is chosen by D and is given as part of the query, similarly for z when D queries $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$. In these situations, since we cannot invert c and z to obtain the $\mathcal{RO}$ inputs that gen-

erated them, we generate these inputs uniformly at random (recall that when creating a new tuple, all fields of that tuple have to be generated, even if they were not part of the particular query that triggered the tuple to be created), but that does not mean that if these inputs were supplied to a $\mathcal{RO}$, it will return the corresponding c and z that were sent to the simulators by D earlier, this behaviour may lead to internal collisions. Below we analyse all internal collision scenarios corresponding to simulators $S^{\mathcal{F}}_{1,x}, S^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ and their inverse variants (as outlined in failure conditions 1(c) and 2(c) above, respectively). In all scenarios, the simulators receive queries of a particular format (depending on the receiving simulator) from D and creates the new tuple (x, y, c, z, t).

1. $S^{\mathcal{F}}_{1,x}$: simulator $S^{\mathcal{F}}_{1,x}$ receives c as part of its query $(\rightarrow, c, y)$, so if c exists in $\mathcal{T}^K_x$, no new tuple will be created and thus no collision. However, when D sends the query $(\leftarrow, z_i, c_i)$ to $(S^{\mathcal{F}}_{1,x})^{-1}$, the latter has no way to know what value of y_i (in combination of the given c_i) would generate the given z_i, so it generates y_i uniformly at random, but it is likely that $\mathcal{F}(\mathcal{F}(c_i), y_i) \neq z_i$. If D later sends the query $(\rightarrow, c, y)$ to $S^{\mathcal{F}}_{1,x}$, the latter generates z by $z = \mathcal{F}(\mathcal{F}(c), y)$, a collision then occurs if $z = z_i$ while $(c, y) \neq (c_i, y_i)$.

2. $S^{\mathcal{F}}_{1,y}$: simulator $S^{\mathcal{F}}_{1,y}$ receives c as part of its query $(\rightarrow, x, c)$, so if c exists in $\mathcal{T}^K_y$, no new tuple will be created and thus no collision. However, when D sends the query $(\leftarrow, z_i, x_i)$ to $(S^{\mathcal{F}}_{1,y})^{-1}$, the latter has no way to know what value of c_i (in combination of the given x_i) would generate the given z_i, so it generates c_i by $c_i = \mathcal{F}(K, y_i)$, where y_i is generated uniformly at random, but it is likely that $\mathcal{F}(\mathcal{F}(c_i), y_i) \neq z_i$. If D later sends the query $(\rightarrow, x, c)$ to $S^{\mathcal{F}}_{1,y}$, the latter generates z by $z = \mathcal{F}(x, \mathcal{F}(c))$, a collision then occurs if $z = z_i$ while $(x, c) \neq (x_i, c_i)$.

3. $S^{\mathcal{F}}_{1,c}$: When D sends the queries $(\leftarrow, c_i, x_i)$ and $(\rightarrow, K, c_j)$ to $(S^{\mathcal{F}}_{1,c})^{-1}$ and $R^{\mathcal{F}}_{1,c}$, respectively, both $(S^{\mathcal{F}}_{1,c})^{-1}$ and $R^{\mathcal{F}}_{1,c}$ have no way to know what values of x_i, y_i and x_j, y_j would generate the given c_i and c_j, so they generate x_i, y_i and x_j, y_j uniformly at random, but it is likely that $\mathcal{F}(x_i, y_i) \neq c_i$ and $\mathcal{F}(x_j, y_j) \neq c_j$. If D later sends the query $(\rightarrow, x, y)$ to $S^{\mathcal{F}}_{1,c}$, the latter generates c by $c = \mathcal{F}(x, y)$, then a collision occurs if $c = c_i$ or $c = c_j$ while $(x, y) \neq (x_i, y_i)$ or $(x, y) \neq (x_j, y_j)$. Similarly, when D sends the query $(\leftarrow, K, z_i)$ to $(R^{\mathcal{F}}_{1,c})^{-1}$, the latter has no way to know what value of c_i would generate the given z_i, so it generates c_i by $c_i = \mathcal{F}(x_i, y_i)$ (while generating x_i, y_i uniformly at random). If D later sends the query $(\rightarrow, x, y)$ to $S^{\mathcal{F}}_{1,c}$, the latter generates z by $z = \mathcal{F}(K, c)$, where $c = \mathcal{F}(x, y)$, a collision then occurs if $z = z_i$ while $c \neq c_i$.

4. $(S^{\mathcal{F}}_{1,x})^{-1}$: simulator $(S^{\mathcal{F}}_{1,x})^{-1}$ cannot generate any collision because it accepts queries of the format $(\leftarrow, z, c)$ where both c and z are given, so if they exist in $\mathcal{T}^K_x$, no new tuple will be generate and thus no collision.

5. $(S^{\mathcal{F}}_{1,y})^{-1}$: simulator $(S^{\mathcal{F}}_{1,y})^{-1}$ cannot generate collisions in the value of z because it receives it as part of its query $(\leftarrow, z, x)$, so if a given z matches an existing z_i in $\mathcal{T}^K_{1,y}$, no new tuple will be generate and thus no collision. However, when D sends the queries $(\rightarrow, x_i, c_i)$ and $(\leftarrow, K, c_j)$ to $S^{\mathcal{F}}_{1,y}$ and $(R^{\mathcal{F}}_{1,y})^{-1}$, respectively, both $S^{\mathcal{F}}_{1,y}$ and $(R^{\mathcal{F}}_{1,y})^{-1}$ have no way to know what values of y_i and y_j would generate the given c_i and c_j, so they generate y_i and y_j uniformly at random. If D later sends the query $(\leftarrow, z, x)$ to $(S^{\mathcal{F}}_{1,y})^{-1}$, the latter first generates y uniformly at random and then generates c by $c = \mathcal{F}(K, y)$, a collision then occurs if $c = c_i$ or $c = c_j$ while $y \neq y_i$ or $y \neq y_j$.

6. $(S^{\mathcal{F}}_{1,c})^{-1}$: simulator $(S^{\mathcal{F}}_{1,y})^{-1}$ cannot generate collision in the value of c because it receives it as part of its query, so if a given c matches an existing c in $\mathcal{T}^K_c$, no new tuple will be generate and thus no collision. However, when D sends the query $(\leftarrow, K, z_i)$ to $(\mathcal{R}^{\mathcal{F}}_{1,c})^{-1}$, the z_i is given in the query, so there is no way for $(\mathcal{R}^{\mathcal{F}}_{1,c})^{-1}$ to know what value of c_i would generate the given z_i, so it generates c_i by $c_i = \mathcal{F}(x_i, y_i)$, where x_i, y_i are generated uniformly at random. If D later sends $(\leftarrow, c, x)$ to $(S^{\mathcal{F}}_{1,c})^{-1}$, the latter generates z by $z = \mathcal{F}(K, \mathcal{F}(c))$, a collision then occurs if $z = z_i$ while $c \neq c_i$.

Scenarios 1,2 and 3 above are the failure conditions 1(c).i., 1(c).ii. and 1(c).iii., respectively, while scenarios 5 and 6 are the failure conditions 2(c).i. and 2(c).ii., respectively (probability of scenario 4 is 0, as discussed above). Failure conditions 1(c).ii. and 1(c).iii. may occur with probability $\leq (q_2 + q_3)^2/2^{m+n}$; in this case, we should consider the collision probability of a *combination* of $x \in \{0,1\}^m$ and $c \in \{0,1\}^n$ (in the case of 1(c).ii.) or $x \in \{0,1\}^m$ and $y \in \{0,1\}^n$ (in the case of 1(c).iii.) since it is the combination what causes the collision not the individual instances of x, y or c, x. Following similar argument, failure condition 1(c).i. occur with probability $\leq (q_2+q_3)^2/2^{2n}$ as the colliding strings in this case are both of length n-bits, that is, we should consider the collision probability of a combination of $c \in \{0,1\}^n$ and $y \in \{0,1\}^n$. In all the failure conditions of 1(c), the collision occurs by two colliding strings, each consists of two values, where the combination of these two values in each string is what causes the collision; this is discussed above and reflected on their probabilities. On the other hand, in failure conditions 2(c).i. and 2(c).ii., the two colliding strings consist of a single value each ($y \in \{0,1\}^n$ in the case of 2(c).i. and $c \in \{0,1\}^n$ in the case of 2(c).ii.). Thus, both failure conditions 2(c).i. and 2(c).ii. may occur with probability $\leq (q_2 + q_3)/2^{2n}$.

Finally, failure condition 2(d) covers collisions caused by partially matched queries. For example, if a simulator received a 2-string query (x, y), it first searches its corresponding table for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $(x, y) = (x_i, y_i)$, here both x and y should match x_i and y_i, respectively. However, if one of these two strings (either x or y) was a match (with x_i or y_i), the simulator will ignore it and proceed to generate a new tuple. This is not an issue if the simulator received a forward query because forward queries generate either c or z, so there will be no collision as along as one of the received strings is distinct (unless one of the other failure conditions is satisfied). On the other hand, this becomes problematic when the simulators receive inverse queries because in this case they will receive c or z as part of the query and this may lead to a collision. We now consider all the inverse simulators $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (S^{\mathcal{F}}_{1,c})^{-1}$ and show how such collisions can occur:

1. $(S^{\mathcal{F}}_{1,x})^{-1}$: when simulator $(S^{\mathcal{F}}_{1,x})^{-1}$ receives the query (z, c), if there is tuple $(x_i, y_i, c_i, z_i, t_i) \in \mathcal{T}^K_x$, then no new tuple will be created. On the other hand, if $c = c_i$ and $z \neq z_i$, a new tuple will be created, but this will not generate a collision since $z \neq z_i$. However, if it is the other way round, $c \neq c_i$ and $z = z_i$, a new tuple will be created, but this time it will cause a collision since $(c, y) \neq (c_i, y_i)$ while $z = z_i$.

2. $(S^{\mathcal{F}}_{1,y})^{-1}$: following the same argument, when simulator $(S^{\mathcal{F}}_{1,y})^{-1}$ receives the query (z, x), new tuple will be created even if there is a tuple $(x_i, y_i, c_i, z_i, t_i) \in \mathcal{T}^K_y$ such that $x \neq x_i$ while $z = z_i$, this will obviously lead to a collision since $(x, c) \neq (x_i, c_i)$ while $z = z_i$.

3. $(S^{\mathcal{F}}_{1,c})^{-1}$: with simulator $(S^{\mathcal{F}}_{1,c})^{-1}$, partial query collision leads to a collision in the c value, but this obviously results in a collision with the z value since, according to $(S^{\mathcal{F}}_{1,c})^{-1}$ definition, $z \leftarrow \mathcal{F}(K, \mathcal{F}(c))$, so as long as there is a collision in c, there will also be a collision in z.

All the scenarios in failure condition 2(d) occur with probably bounded by the birthday attack. In 2(d).i. the collision occurs due to $c \in \{0,1\}^n$, thus the probably is bound by $(q_2 + q_3)^2/2^n$. On the other hand, in failure conditions 2(d).ii. and 2(d).iii., the collisions occur due to $x \in \{0,1\}^m$, so the probability is bounded by $(q_2 + q_3)^2/2^m$. Note that partial query collisions cannot occur with simulators $R^{\mathcal{F}}_{1,i}$ and $(R^{\mathcal{F}}_{1,i})^{-1}$ (where $i \in \{x, y, c\}$) because these simulators accept a single string (in addition to the fixed key K), so if it matches one of the existing tuple, it will be detected immediately.

Other than the failure conditions above, we prove that collisions in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ cannot occur. As there are two types of tuples in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ (sequenced and singular), there are four possible collision scenarios, namely: collisions between tuples

from different sequences, collisions among tuples within the same sequence, collisions among singular tuples, and collisions between sequenced and singular tuples; these are covered by lemmas 5.4.2, 5.4.4, 5.4.6 and 5.4.7, respectively. These proofs apply for both the modified S simulators in this game, and the modified R simulators in $G(5)$; that is, in $G(5)$ we will modify the simulators $R_x^{\mathcal{F}}, R_y^{\mathcal{F}}, R_c^{\mathcal{F}}$ (and their inverse variants) and introduce failure conditions similar to the ones we introduced in this game, then the distinguishing probability of $G(5)$ will be the success probability of the failure conditions there, other than those failure conditions, the following lemmas prove that $R_{1,x}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}, R_{1,c}^{\mathcal{F}}$ (and their inverse variants) cannot generate collisions.

Lemma 5.4.2 (Collision freeness among sequences). *For any two sequences, Seq_1 and Seq_2, in a table $\mathcal{T}_i^K$ that is maintained by the simulators $S_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, R_{1,i}^{\mathcal{F}}, (R_{1,i}^{\mathcal{F}})^{-1}$, a tuple in Seq_1 cannot collide with another in Seq_2, where $i \in \{x, y, c\}$.*

Proof. Let s_h and s_f index two different sequences, Seq_1 and Seq_2, in the table $\mathcal{T}_i^K, i \in \{x, y.c\}$, consisting of u and v tuples, respectively, and let Seq_1 be rooted by the i-th tuple while Seq_2 be rooted by the j-th tuple:

$$\begin{aligned} Seq_1 &= (x_i^{s_h,0}, y_i^{s_h,0}, c_i^{s_h,0}, z_i^{s_h,0}, t_i^{s_h,0}) \ldots (x_{\_}^{s_h,u}, y_{\_}^{s_h,u}, c_{\_}^{s_h,u}, z_{\_}^{s_h,u}, t_{\_}^{s_h,u}) \\ Seq_2 &= (x_j^{s_f,0}, y_j^{s_f,0}, c_j^{s_f,0}, z_j^{s_f,0}, t_j^{s_f,0}) \ldots (x_{\_}^{s_f,v}, y_{\_}^{s_f,v}, c_{\_}^{s_f,v}, z_{\_}^{s_f,v}, t_{\_}^{s_f,v}) \end{aligned}$$

where $y_i^{s_h,0} = y_j^{s_f,0} = IV$ and $h \neq f$. A collision between tuple $(x_a, y_a, c_a, z_a, t_a)$ in Seq_1 and tuple $(x_b, y_b, c_b, z_b, t_b)$ in Seq_2 occurs when $z_a = z_b$ while $(x_a, y_a, c_a) \neq (x_b, y_b, c_b)$. Thus, we need to show that whenever a sequenced tuple is created, it cannot collide with any other existing sequenced tuple. Here our discussion is based on the assumption that the z value of a tuple is being generated by the simulators, but in simulators $(S_{1,x}^{\mathcal{F}})^{-1}, (S_{1,y}^{\mathcal{F}})^{-1}, (R_{1,c}^{\mathcal{F}})^{-1}$, the z value is given to the simulators as part of the queries. These simulators, however, will only generate singular tuples while here we are only concerned with sequenced tuples. In fact, the only simulators that will generate sequences are $S_{1,x}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$, so it suffices to investigate how these simulators generate their z values because if there is a collision, they are the only ones that could have cause it. The easiest way to do this is to observe the input that the simulators use to generate their z values and prove that z will be unique as long as these inputs are unique (generated in a collision-free manner).

Simulator $S_{1,x}^{\mathcal{F}}$ generates its z by $\mathcal{F}(\mathcal{F}(c), y)$, simulator $R_{1,y}^{\mathcal{F}}$ generates its z by $\mathcal{F}(T||x, c)$, and simulator $S_{1,c}^{\mathcal{F}}$ generates its z by $\mathcal{F}(K, T||x)$, where T is a sequence of x's. We now show that the inputs $(\mathcal{F}(c), y), (T||x, c), (K, T||x)$ are prepared in a collision-free manner (i.e., they are unique) which will immediately imply that the z values they will generate are also collision-free since the latter is generated by a $\mathcal{RO}$.

- $S^{\mathcal{F}}_{1,x}(\rightarrow, c, y)$: in this case, the z value is created by the query $(\mathcal{F}(c), y)$ to the random oracle $\mathcal{F}$, so we show that both c and y are unique. It is clear that here the y value cannot collide with any existing y value because the simulator receives y as part of its query. If there is an existing tuple $(x_i, y_i, c_i, z_i, t_i) \in \mathcal{T}^K_x$ such that $y_i = y$, then $S^{\mathcal{F}}_{1,x}$ will only create a new tuple if $c \neq c_i$, otherwise if $c = c_i$, then $(c, y) = (c_i, y_i)$ and a new tuple will not be created. This also includes the case when $y = IV$, where a new tuple will only be created if $c \neq c_i$. This mean that the $S^{\mathcal{F}}_{1,x}$ simulator guarantees a total collision freeness, which implies that $\mathcal{F}(\mathcal{F}(c), y) \neq \mathcal{F}(\mathcal{F}(c_i), y_i)$ will hold for any $(c_i, y_i) \in \mathcal{T}^K_x$, which immediately implies $z_i \neq z_j$.

- $R^{\mathcal{F}}_{1,y}(\rightarrow, K, y)$: this simulator creates its z value by querying $\mathcal{F}$ with $(T||x, c)$, where T is a sequence of the x's of the preceding tuples in the sequence to which the new query belongs. It is easy to see that T is unique for a particular z because this sequence of x's will only occur for that particular query. On the other hand, c here is being generated by $\mathcal{F}(K, y)$, which implies that as long as $y \neq y_i$ for some $y_i \in \mathcal{T}^K_y$, then $\mathcal{F}(K, y_i) \neq \mathcal{F}(K, y_j)$ holds, which, in turn, implies $c_i \neq c_j$, and this will always be the case since y here is received as part of the query and if it matches any $y_i \in \mathcal{T}^K_y$, no new tuple will be created.

- $S^{\mathcal{F}}_{1,c}(\rightarrow, x, y)$: in this simulator, the z value is created by querying $(K, T||x)$ to the random oracle $\mathcal{F}$, where K is a fixed key and T is a sequence of x's. In this case, the collision freeness of z solely depends on the value T, which, as discussed above, is unique for any particular query.

Finally, it is easy to see that these results will also generalise to cases when there are multiple sequences in $\mathcal{T}^K_i$ where $i \in \{x, y, c\}$. □

Corollary 5.4.3 (Prefix collision freeness among sequences). *If any of the tables $\mathcal{T}^K_i$ maintained by the simulators $S^{\mathcal{F}}_{1,i}, (S^{\mathcal{F}}_{1,i})^{-1}, R^{\mathcal{F}}_{1,i}, (R^{\mathcal{F}}_{1,i})^{-1}$, where $i \in \{x, y, c\}$, contains two different sequences Seq_1 and Seq_2, then any tuple in Seq_1 cannot be a prefix of any tuple in Seq_2, and vice versa.*

Proof. The proof follows from lemma 5.4.2. A tuple $(x^{s_h,p}_a, y^{s_h,p}_a, c^{s_h,p}_a, z^{s_h,p}_a, t^{s_h,p}_a)$ in Seq_1 cannot be a prefix of another tuple $(x^{s_f,q}_b, y^{s_f,q}_b, c^{s_f,q}_b, z^{s_f,q}_b, t^{s_f,q}_b)$ in Seq_2, that is $z^{s_h,p}_a$ cannot equal to $y^{s_f,q}_b$, because if that was the case, then this is merely a collision between $z^{s_h,p}_a$ and $z^{s_f,q-1}_{\_}$ since $y^{s_f,q}_b = z^{s_f,q-1}_{\_}$, and as shown in lemma 5.4.2, a collision cannot occur in this case. Clearly, the other way round also holds; no tuple from Seq_2 may be a prefix to another in Seq_1. Also, a sequence cannot be a prefix of itself. That

is, given a sequence Seq_3 rooted at the i-th tuple and has a tail at the j-th tuple, then $y_i \neq z_j$ holds since $y_i = IV$. However, with probability $1/2^n$ (where $n = |z_j|$), it might be the case that $z_j = IV$, but this is covered in failure conditions 1(a) and 2(a), which also covers prefix collisions within a single sequenced tuple. □

Lemma 5.4.4 (Collision freeness within a single sequence). *If any of the tables $\mathcal{T}_i^K$ maintained by the simulators $S_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, R_{1,i}^{\mathcal{F}}, (R_{1,i}^{\mathcal{F}})^{-1}$, where $i \in \{x, y, c\}$, contains a sequence Seq_π, then any tuple in that sequence cannot collide with any other tuple in the same sequence.*

Proof. Recall that the only simulators generating sequences are $S_{1,x}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$. Let $Seq_\pi = (x_a^{s_l,0}, y_a^{s_l,0} = IV, c_a^{s_l,0}, z_a^{s_l,0}, t_a^{s_l,0}) \dots (x_{\_}^{s_l,n}, y_{\_}^{s_l,n}, c_{\_}^{s_l,n}, z_{\_}^{s_l,n}, t_{\_}^{s_l,n})$ be a sequence containing $n > 1$ tuples and rooted by the a-th tuple. An internal collision in the sequence s_l implies that there exists $z_{\_}^{s_l,i} = z_{\_}^{s_l,j}$, where $i \neq j$ and $i, j \in \{0, \dots, n\}$. In $R_{1,y}^{\mathcal{F}}$ and $S_{1,c}^{\mathcal{F}}$, this happens if the following equality holds:

$$x_{\_}^{s_l,0}||x_{\_}^{s_l,1}|| \dots ||x_{\_}^{s_l,i-1}||x_{\_}^{s_l,i} = x_{\_}^{s_l,0}||x_{\_}^{s_l,1}|| \dots ||x_{\_}^{s_l,j-1}||x_{\_}^{s_l,j}$$

Since z is being generated by the random oracle $\mathcal{F}$, this is only possible if $i = j$. On the other hand, in $S_{1,x}^{\mathcal{F}}$, an internal collision happens between two tuples $(x_p, y_p, c_p, z_p, t_p)$ and $(x_q, y_q, c_q, z_q, t_q)$ belonging to the same sequence if $z_p = z_q$. This can happen only if $(c_p, y_p) = (c_q, y_q)$, which is not possible since both c and y are given to $S_{1,x}^{\mathcal{F}}$ as part of the query and would only cause a new tuple to be created if there is no existing tuple matching the queried c and y. □

Corollary 5.4.5 (Ancestors and descendants of sequenced tuples). *In any of the tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, a single tuple cannot be part of more than one sequence. More generally, a single tuple in $\mathcal{T}_i^K, i \in \{x, y, c\}$ cannot have more than one descendant tuple and more than one parent tuple.*

Proof. The proof follows from lemmas 5.4.2 and 5.4.4. Let the i-th tuple indexed by s_l, p be the parent of the tuple indexed by $s_l, p+1$ of the sequence l, that is $z_i^{s_l,p} = y_{\_}^{s_l,p+1}$. The only way the tuple s_l, p can have another descendant s_l, k is when $z_i^{s_l,p} = y_{\_}^{s_i,k}$, which implies that $y_{\_}^{s_i,p+1} = y_{\_}^{s_i,k}$, but this cannot happen because collisions and prefixes cannot occur as shown in lemmas 5.4.2 and 5.4.4. Similarly, and following the same argument, tuple s_l, p with parent $s_l, p-1$ cannot have another parent s_l, k' because this implies $z_{\_}^{s_l,p-1} = z_{\_}^{s_l,k'} = y_i^{s_l,p}$, which cannot occur. □

Lemma 5.4.6 (Collision freeness among singular tuples). *In any of the tables $\mathcal{T}_i^K$ maintained by the simulators $S_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, R_{1,i}^{\mathcal{F}}, (R_{1,i}^{\mathcal{F}})^{-1}$, where $i \in \{x, y, c\}$, collisions cannot occur between two singular tuples within the same table.*

Proof. A collision in this context means there are two tuples $(x_i, y_i, c_i, z_i, t_i = \perp)$ and $(x_j, y_j, c_j, z_j, t_j = \perp)$, such that $z_i = z_j$ while $(x_i, y_i, c_i) \neq (x_j, y_j, c_j)$. However, since z is always being generated by the random oracle $\mathcal{F}$ (unless given by D), $z_i = z_j$ will only hold if the inputs given to $\mathcal{F}$ to generate z_i and z_j are identical. In lemma 5.4.2 we considered how sequenced tuples are being generated by $S_{1,x}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}$, in this proof, we need to consider the other simulators which generate singular tuples (in addition to $S_{1,x}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}$ since these can also generate singular tuples), these are:

$$(S_{1,x}^{\mathcal{F}})^{-1}, S_{1,y}^{\mathcal{F}}, (S_{1,y}^{\mathcal{F}})^{-1}, (S_{1,c}^{\mathcal{F}})^{-1}, (R_{1,y}^{\mathcal{F}})^{-1}, R_{1,x}^{\mathcal{F}}, (R_{1,x}^{\mathcal{F}})^{-1}, R_{1,c}^{\mathcal{F}}, (R_{1,c}^{\mathcal{F}})^{-1}$$

In all these cases, unless given by D, the value of z is generated based on c, so it only suffices to prove that in each case, the way c is being generated is collision-free to imply that z will be collision-free. Precisely, z_i of a singular tuple can either by generated by $z_i = \mathcal{F}(X, c_i)$ or $z = \mathcal{F}(X, \mathcal{F}(c_i))$, where $X \in \{K, x_i, y_i\}$. Therefore, we now have two possible inputs to z, making four possible collision scenarios:

- $z_i = \mathcal{F}(X, c_i), z_j = \mathcal{F}(X, c_j)$: the only simulator that generates $z_i = \mathcal{F}(X, c_i)$ is $R_{1,x}^{\mathcal{F}}$, so we show that $R_{1,x}^{\mathcal{F}}$ cannot generate $c_i = c_j$ while $i \neq j$. It is easy to see that this is always the case since $R_{1,x}^{\mathcal{F}}$ uses $\mathcal{F}$ to generate $c_i = \mathcal{F}(K, x_i), c_j = \mathcal{F}(K, x_j)$, so as long as $(K, x_i) \neq (K, x_j)$, then $c_i \neq c_j$ will hold, which is always the case since $R_{1,x}^{\mathcal{F}}$ receives x_i in the query and will only use it to create a new tuple if there is no $x_j \in \mathcal{T}_x^K$ such that $x_i = x_j$.

- $z_i = \mathcal{F}(X, \mathcal{F}(c_i)), z_j = \mathcal{F}(X, \mathcal{F}(c_j))$: apart from $R_{1,x}^{\mathcal{F}}$, all other simulators that generate singular tuples generate z as $\mathcal{F}(X, \mathcal{F}(c_i))$, so we only need to show that $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$ will always hold if $c_i \neq c_j$. In all simulators generating singular tuples, c_i is given in the query, so upon receiving a query (X, c_i), the simulators first check if there is a tuple $(x_j, y_j, c_j, z_j, t_j) \in \mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ such that $c_i = c_j$, if it does, they do not create a new tuple. Thus, as long as $c_i \notin \mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, then $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$ follows implying $z_i \neq z_j$. An even simpler argument is that $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$ trivially holds as long as $c_i \neq c_j$ because $\mathcal{F}$ is a random oracle.

- $z_i = \mathcal{F}(X, \mathcal{F}(c_i)), z_j = \mathcal{F}(X, c_j)$ or $z_i = \mathcal{F}(X, c_i), z_j = \mathcal{F}(X, \mathcal{F}(c_j))$: here we only need to show that $\mathcal{F}(c_i) \neq c_j$, where $i \neq j$, to imply $\mathcal{F}(X, \mathcal{F}(c_i)) \neq \mathcal{F}(X, c_j)$. We know that c_j in $\mathcal{F}(X, c_j)$ is being generated by $\mathcal{F}(x_j, y_j)$ or $\mathcal{F}(K, x_j)$ or

$\mathcal{F}(K, y_j)$. Thus, we have $\mathcal{F}(c_i) = \mathcal{F}(x_j, y_j)$ or $\mathcal{F}(c_i) = \mathcal{F}(K, x_j)$ or $\mathcal{F}(c_i) = \mathcal{F}(K, y_j)$, none of which can hold since $\{0,1\}^n \neq [\{0,1\}^m||\{0,1\}^n], \{0,1\}^n \neq [\{0,1\}^k||\{0,1\}^m], \{0,1\}^n \neq [\{0,1\}^k||\{0,1\}^n]$, respectively.

It remains to discuss the case when z is given in the query, which can happen only with $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$, but it is easy to see that $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$ will not generate collisions because, upon a new query, they will first check $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$, respectively, for any tuple with a similar z and would only create a new tuple if no such z exists. This covers all possible collision scenarios among singular tuples. □

Lemma 5.4.7 (Collision freeness between singular and sequenced tuples). *If any of the tables $\mathcal{T}^K_i$ maintained by the simulators $S^{\mathcal{F}}_{1,i}, (S^{\mathcal{F}}_{1,i})^{-1}, R^{\mathcal{F}}_{1,i}, (R^{\mathcal{F}}_{1,i})^{-1}$, where $i \in \{x, y, c\}$, contains a sequence Seq_π, then no singular tuple within the same table may collide with any tuple in Seq_π.*

Proof. Here we have two cases: either the newly generated tuple $(x_i, y_i, c_i, z_i, t_i)$ is singular while colliding with an existing sequenced tuple $(x_j, y_j, c_j, z_j, t_j)$, or the other way round (collision implies $z_i = z_j$). Either way, according to the definition of simulators $S^{\mathcal{F}}_{1,k}, R^{\mathcal{F}}_{1,k}, k \in \{x, y, c\}$, regardless of whether it was singular or sequenced tuple, z will always be generated by $\mathcal{F}$ (unless it is given by D). Let $(x_i, y_i, c_i, z_i, t_i = \perp)$ be a singular tuple while $(x_j^{s_l,p}, y_j^{s_l,p}, c_j^{s_l,p}, z_j^{s_l,p}, t_j^{s_l,p} = 1)$ be a sequenced tuple, with $p \geq 1$. In this case, $z_i = \mathcal{F}(X, c_i)$ or $z_i = \mathcal{F}(X, \mathcal{F}(c_i))$, where $X \in \{K, x_i, y_i\}$, while $z_j = \mathcal{F}(K, x_{\_}^{s_l,0}||x_{\_}^{s_l,1}|| \ldots ||x_{\_}^{s_l,p-1}||x_{\_}^{s_l,p})$ when generated by $S^{\mathcal{F}}_{1,c}$ or $z_j = \mathcal{F}(c_j, x_{\_}^{s_l,0}||x_{\_}^{s_l,1}|| \ldots ||x_{\_}^{s_l,p-1}||x_{\_}^{s_l,p})$ when generated by $R^{\mathcal{F}}_{1,y}$ or $z_j = \mathcal{F}(c_j, y_j)$ when generated by $S^{\mathcal{F}}_{1,x}$. When z_j is generated by $S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_{1,y}$, unless $c_i = x_{\_}^{s_l,0}||x_{\_}^{s_l,1}|| \ldots ||x_{\_}^{s_l,p-1}||x_{\_}^{s_l,p}$, $z_i \neq z_j$ will always hold. Similarly, when z_j is generated by $S^{\mathcal{F}}_{1,x}$, $z_j \neq z_i$ will hold as long as $(c_j, y_i) \neq (c_i, y_i)$ also holds, which is always the case as such collision would have been detected by $S^{\mathcal{F}}_{1,x}$ before creating the j-th or i-th tuple (whichever queried last). It is also possible that the sequence l contains only one tuple, that is, only the root tuple $(x_j^{s_l,0}, y_j^{s_l,0} = IV, c_j^{s_l,0}, z_j^{s_l,0}, t_j^{s_l,0})$, in which case a collision occurs if $z_i = z_j^{s_l,0}$, which is not possible as long as $c_i \neq c_j^{s_l,0}$ since both z_i, z_j are generated by $\mathcal{F}$ and c is always involved in their generation process. According to the definition of the simulators $S^{\mathcal{F}}_{1,k}, R^{\mathcal{F}}_{1,k}, k \in \{x, y, c\}$, unless c is given by D, it will be generated by $\mathcal{F}$. If c is sequenced it is generated by $\mathcal{F}(x, y)$ or $\mathcal{F}(K, y)$, otherwise if c is singular it is generated by $\mathcal{F}(K, y)$ or $\mathcal{F}(K, x)$ or $\mathcal{F}(x, y)$. It is easy to see that the fact that $y_j^{s_l,0} = IV$ while $y_i \neq IV$ always holds (otherwise y_i would not be singular) implies that the following

also hold (and thus any $c_j \neq c_i$).

$$\Big[\mathcal{F}(x,y) \neq \mathcal{F}(K,y)\Big], \Big[\mathcal{F}(x,y) \neq \mathcal{F}(K,x)\Big], \Big[\mathcal{F}(K,y) \neq \mathcal{F}(K,x)\Big]$$

To complete the proof, it remains to investigate cases when c is given in a query. In all simulators generating singular tuples, when c is given by D, c is not directly used to generate z, rather, $z_i = \mathcal{F}(X, \mathcal{F}(c_i)), X \in \{x_i, y_i\}$, so a collision between a singular tuple and a sequenced one implies $\mathcal{F}(c_i) = c_j^{s_l,0}$ should hold[3], which, for the case of $S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_{1,y}$, can be rewritten as $\mathcal{F}(c_i) = \mathcal{F}(x_j^{s_l,p}||y_j^{s_l,p})$ and $\mathcal{F}(c_i) = \mathcal{F}(K||y_j^{s_l,p})$, but clearly $c_i \neq x_j^{s_l,p}||y_j^{s_l,p}$ and $c_i \neq K||y_j^{s_l,p}$, since $\{0,1\}^n \neq \{0,1\}^m||\{0,1\}^n$ and $\{0,1\}^n \neq \{0,1\}^k||\{0,1\}^n$, respectively. For the case when c_j is generated by $S^{\mathcal{F}}_{1,x}$, $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$ still holds because $\mathcal{F}(c_i) \neq \mathcal{F}(\mathcal{F}(c_j), y_j)$ since $\{0,1\}^n \neq \{0,1\}^n||\{0,1\}^n$. Finally, it is trivial to see that $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, R^{\mathcal{F}}_{1,c})^{-1}$ will not generate collisions, even though they accepts z from D, because if a queried z already exists in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ no new (colliding) tuple will be created. □

Finally, the probability of this game is the sum of the probabilities of the failure conditions of $S^{\mathcal{F}}_{1,x}, S^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ (and their inverse variants):

$$\begin{aligned}
P_3^x &= \Pr[D^{F_1^{\mathcal{F}}, S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_x} = 1] \\
&\leq \big(2^n((q_2+q_3)+(q_2+q_3)^2)+(q_2+q_3)\big)/2^{2n} \\
P_3^y &= \Pr[D^{F_1^{\mathcal{F}}, S^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_y} = 1] \\
&\leq \big((q_2+q_3)(2^{m+1}(q_2+q_3)+2^{m+1}+2^n(q_2+q_3)+1)\big)/2^{m+n} \\
P_3^c &= \Pr[D^{F_1^{\mathcal{F}}, S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_c} = 1] \\
&\leq \big((q_2+q_3)(3\cdot 2^m+2^{m+1}(q_2+q_3)+2^n(q_2+q_3)+1))\big)/2^{m+n}
\end{aligned}$$

Game 4. Similar to $G(3)$, in this game we introduce slightly modified replicas of the simulators $R_x^{\mathcal{F}}, R_y^{\mathcal{F}}, R_c^{\mathcal{F}}$, still with oracle access to $\mathcal{F}$. The distinguisher D will now interact with the modified simulators $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$ which, upon receiving a query, create the new tuple (x,y,c,z,t) but explicitly abort if any of the following failure conditions is satisfied:

1. On forward queries $R^{\mathcal{F}}_{1,x}(\rightarrow,K,x), R^{\mathcal{F}}_{1,y}(\rightarrow,K,y), R^{\mathcal{F}}_{1,c}(\rightarrow,K,c)$, the simulators create the new tuple (x,y,c,z,t), but the following collisions occur:

   1(a). Fixed point: in $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$, it is the case that $z = IV$, and in $R^{\mathcal{F}}_{1,y}$, it is also the case that $z = y$.

[3]Recall that the only simulators generating sequences are $S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ and that $R^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ generate c by $\mathcal{F}(K,y), \mathcal{F}(x,y)$, respectively, while $S^{\mathcal{F}}_{1,x}$ receives c as part of its queries.

1(b). Prefix collision: in $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$, it is the case that $z = y_j$ for some $y_j \in \mathcal{T}^K_x(\hat{y}) \cup \mathcal{T}^K_y(\hat{y}) \cup \mathcal{T}^K_c(\hat{y})$.

1(c). Internal collision: there is a tuple $(x_i, y_i, c_i, z_i, t_i)$ in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ such that:

i. when $R^{\mathcal{F}}_{1,x}$ receives $(\rightarrow, K, x)$, the following hold: $c_i = c \wedge x_i \neq x$.

ii. when $R^{\mathcal{F}}_{1,y}$ receives $(\rightarrow, K, y)$, the following hold: $c_i = c \wedge y_i \neq y$.

2. On inverse queries $(R^{\mathcal{F}}_{1,x})^{-1}(\leftarrow, K, c), (R^{\mathcal{F}}_{1,y})^{-1}(\leftarrow, K, c), (R^{\mathcal{F}}_{1,c})^{-1}(\leftarrow, K, z)$, the simulators create the new tuple (x, y, c, z, t), but the following collisions occur:

2(a). Fixed point: in $(R^{\mathcal{F}}_{1,x})^{-1}, (R^{\mathcal{F}}_{1,y})^{-1}$, it is the case that $z = IV$.

2(b). Prefix collision: in $(R^{\mathcal{F}}_{1,x})^{-1}, (R^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$, it is the case that $z = y_j$ for some $y_j \in \mathcal{T}^K_x(\hat{y}) \cup \mathcal{T}^K_y(\hat{y})$.

2(c). Internal collision: we show that $(R^{\mathcal{F}}_{1,x})^{-1}, (R^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$ cannot generate collisions, see below.

As illustrated in $G(3)$, failure conditions 1(a) and 2(a) occur with probability $(q_2 + q_3)/2^n$ each. Simulators $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$ are susceptible to the first fixed point type (when $z = IV$), while only simulator $R^{\mathcal{F}}_{1,y}$ is susceptible to the second fixed point type (when $z = y_i$), in the latter case simulators $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,c}$ are not susceptible to the second fixed point type because they generate their y value uniformly at random excluding all existing values of z and y in tables $\mathcal{T}^K_x, \mathcal{T}^K_c$. Thus, failure condition 1(a) occurs for simulators $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,c}$ with probability $(q_2 + q_3)/2^n$, while it occurs for simulator $R^{\mathcal{F}}_{1,y}$ with probability $2(q_2 + q_3)^2/2^n$. Similar analysis apply for failure condition 2(a).

Following the same argument in $G(3)$, both failure conditions 1(b) and 2(b) are at most the birthday bound $(q_2 + q_3)^2/2^n$ and are applicable to all simulators in this game. In particular, all simulators except $(R^{\mathcal{F}}_{1,c})^{-1}$ generate their z values by querying a $\mathcal{RO}$, meaning that they cannot force the $\mathcal{RO}$ to exclude the existing y values in the tables $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ from the newly generated z value. In the case of $(R^{\mathcal{F}}_{1,c})^{-1}$, z is given as part of the query and the simulator only checks whether there is a tuple $(x_i, y_i, c_i, z_i, t_i) \in \mathcal{T}^K_c$ such that $z = z_i$ before creating a new tuple, it, however, does not check for $z = y_i$, so this prefix collision can still happen with probability $(q_2 + q_3)^2/2^n$. We now discuss in details failure conditions 1(c) and 2(c).

1. $R^{\mathcal{F}}_{1,x}$: When D sends the query $(\rightarrow, c, y)$ to $S^{\mathcal{F}}_{1,x}$, the latter has no way to know which x generated the given c, so it generates x uniformly at random, but it is most likely that $\mathcal{F}(K, x) \neq c$. When D later sends the query $(\rightarrow, K, x_i)$ to $R^{\mathcal{F}}_{1,x}$, the latter generates $c_i = (K, x_i)$. A collision occurs if $c_i = c \wedge x_i \neq x$.

2. $R^{\mathcal{F}}_{1,y}$: When D sends the query $(\rightarrow, x, c)$ to $S^{\mathcal{F}}_{1,y}$, the latter has no way to know which y generated the given c, so it generates y uniformly at random, but it is most likely that $\mathcal{F}(K, x) \neq c$. When D later sends the query $(\rightarrow, K, y_i)$ to $R^{\mathcal{F}}_{1,y}$, the latter generates $c_i = (K, x_i)$. A collision occurs if $c_i = c \wedge y_i \neq y$.

3. Simulators $R^{\mathcal{F}}_{1,c}$, $(R^{\mathcal{F}}_{1,x})^{-1}$, $(R^{\mathcal{F}}_{1,y})^{-1}$, $(R^{\mathcal{F}}_{1,c})^{-1}$ do not generate any collision because the queries made to them by D contain either c or z values, which if they match any of the existing c or z values in $\mathcal{T}^K_x$ or $\mathcal{T}^K_y$ or $\mathcal{T}^K_c$, no new tuple is generate and thus no collision.

Scenarios 1 and 2 are are failure conditions 1(c).i. and 2(c).ii., respectively. Failure condition 1(c).i. may occur with probability $\leq (q_2 + q_3)/2^m$ since the probably is taken over range size of 2^m (i.e., $x \in \{0,1\}^m$). In this case, x is generated uniformly at random and is assumed to have generated the c value that was sent the query $(\rightarrow, c, y)$, but when later D sends a query with x_i (i.e. $(\rightarrow, K, x_i)$) and it turned out that x_i is the value that really generates the previously sent c when given to a $\mathcal{RO}$, then x and x_i collide at c, this happens with probability $1/2^m$ since there is only one x value generating a particular c; after $q_2 + q_3$ queries, the probability is $(q_2 + q_3)/2^m$. Following similar argument, scenario 2(c).i. may occur with probability $\leq (q_2 + q_3)/2^n$ since the collision occurs between strings of size n-bit, (i.e., $y \in \{0,1\}^n$). Finally, failure condition 2(c) occur with probability 0, as discussed in scenario 3 above. Thus, the final probability of $G(4)$ is:

$$
\begin{aligned}
P^x_4 = \Pr[D^{F^{\mathcal{F}}_1, S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,x}} = 1] &\leq \left(2(q_2+q_3) + 2(q_2+q_3)^2\right)/2^n + (q_2+q_3)/2^m \\
P^y_4 = \Pr[D^{F^{\mathcal{F}}_1, S^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,y}} = 1] &\leq \left(3(q_2+q_3) + 3(q_2+q_3)^2\right)/2^n \\
P^c_4 = \Pr[D^{F^{\mathcal{F}}_1, S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_{1,c}} = 1] &\leq \left(q_2+q_3 + 2(q_2+q_3)^2\right)/2^n
\end{aligned}
$$

Game 5. In this game, we modify the relay algorithms $F^{\mathcal{F}}_{1,x}, F^{\mathcal{F}}_{1,y}, F^{\mathcal{F}}_{1,c}$ to make them dependant on $(S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,x}), (S^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}), (S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_{1,c})$ instead of $\mathcal{F}$, and thus simulating the constructions $\hat{C}^{\mathcal{H}_x,\mathcal{G}_x}_x, \hat{C}^{\mathcal{H}_y,\mathcal{G}_y}_y, \hat{C}^{\mathcal{H}_c,\mathcal{G}_c}_c$, respectively (as defined in figure 5-1). We prove that $F^{\mathcal{F}}_{1,x}, F^{\mathcal{F}}_{1,y}, F^{\mathcal{F}}_{1,c}$ and $F^{S_{1,x},R_{1,x}}_{2,x}, F^{S_{1,y},R_{1,y}}_{2,y}, F^{S_{1,c},R_{1,c}}_{2,c}$ (the modified relay algorithms) behave consistently as long as the sequenced tuples in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ are preserved.

Lemma 5.4.8 (Indistinguishability of G(5)). *The modified relay algorithms introduced in G(5), $F^{S_{1,x},R_{1,x}}_{2,x}, F^{S_{1,y},R_{1,y}}_{2,y}, F^{S_{1,c},R_{1,c}}_{2,c}$, with access to the simulators $(S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,x})$, $(S^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}), (S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_{1,c})$, respectively, are either indistinguishable or behave consistently with $F^{\mathcal{F}}_{1,x}, F^{\mathcal{F}}_{1,y}, F^{\mathcal{F}}_{1,c}$, relay algorithms with oracle access to a random oracle $\mathcal{F}$.*

Proof. Let $X = x_1||x_2||\dots||x_n$ be a message consisting of n m-bit blocks, and let K be a fixed k-bit key. When X is given as a query, $F^{\mathcal{F}}_{1,c}$ processes it as a whole by sending it to the random oracle $\mathcal{F}$, that is $F^{\mathcal{F}}_{1,i}(K, X) = \mathcal{F}(K||x_1||x_2||\dots||x_n)$ and then $z \in \{0,1\}^n$ is obtained, where $K \in \{0,1\}^k$ is a key and $i \in \{x, y, c\}$. First, we prove that $F^{\mathcal{F}}_{1,x}(K, X)$ is indistinguishable from $F^{S_{1,x},R_{1,x}}_{2,x}(K, X)$. Indistinguishability here means that the distinguisher D cannot distinguish between responses from $F^{\mathcal{F}}_{1,x}$ and $F^{S_{1,x},R_{1,x}}_{2,x}$ (except with negligible probability), but not necessarily require the two responses to be identical for similar queries. In fact, $F^{\mathcal{F}}_{1,x}(K, X) = F^{S_{1,x},R_{1,x}}_{2,x}(K, X)$ never holds. To see why, suppose $X = x_1 \in \{0,1\}^m$ and $K \in \{0,1\}^k$, now $F^{\mathcal{F}}_{1,x}(K, x_1) = \mathcal{F}(K||x_1) = v_1$ while $F^{S_{1,x},R_{1,x}}_{2,x}(K, x_1) = S^{\mathcal{F}}_{1,x}(Y, R^{\mathcal{F}}_{1,x}(K, x_1)) = \mathcal{F}(Y, \mathcal{F}(K, x_1)) = v_2$ for some $Y \in \{0, 1, \}^n$. Clearly, $v_1 \neq v_2$ always holds since

$$\Big[\mathcal{F}(K||x_1) = \mathcal{F}(\{0,1\}^{k+m})\Big] \neq \Big[\mathcal{F}(K||\mathcal{F}(y_1||x_1)) = \mathcal{F}(\{0,1\}^{k+n})\Big]$$

This will also apply when $X = x_1, x_2, \dots, x_n$. Furthermore, in $F^{S_{1,x},R_{1,x}}_{2,x}$, getting every input block x_i preprocessed by $R^{\mathcal{F}}_{1,x}$ thwarts other distinguishing attacks. Next, we prove $F^{\mathcal{F}}_{1,c}(K, X) = F^{S_{1,c},R_{1,c}}_{2,c}(K, X)$, the proof of $F^{\mathcal{F}}_{1,y}(K, X) = F^{S_{1,y},R_{1,y}}_{2,y}(K, X)$ is similar. Unlike $F^{\mathcal{F}}_{1,c}$, $F^{S_{1,c},R_{1,c}}_{2,c}$ processes an incoming query by first partitioning it into blocks and then processes each block separately through $S^{\mathcal{F}}_{1,c}$ and $R^{\mathcal{F}}_{1,c}$. Formally, when $F^{S_{1,c},R_{1,c}}_{2,c}$ receives the query (K, X), it beings by dividing X into $x_1, x_2, \dots, x_n$ and then querying $S^{\mathcal{F}}_{1,c}(IV, x_1)$. Once $S^{\mathcal{F}}_{1,c}$ receives this query it immediately creates a sequence in $\mathcal{T}^K_c$ rooted with the tuple $(x_i = x_1, y_i = IV, c_i, z_i, t_i = 0)$ where c_i and z_i are obtained based on the definition of $S^{\mathcal{F}}_{1,c}$. The simulator $S^{\mathcal{F}}_{1,c}$ will then return c_i to $F^{S_{1,c},R_{1,c}}_{2,c}$ which will immediately send it to $R^{\mathcal{F}}_{1,c}(K, c_i)$ and eventually gets z_i. At this stage $F^{S_{1,c},R_{1,c}}_{2,c}$ has completed processing the first block x_1, so it proceeds to process the second block $S^{\mathcal{F}}_{1,c}(z_i, x_2)$. Once $S^{\mathcal{F}}_{1,c}$ receives this new query, it detects it as a sequenced tuple and links the new tuple $(x_j = x_2, y = z_i, c_j, z_j, t_j = 1)$ to the root tuple (the i-th tuple). Note that z_j is not created randomly, instead $S^{\mathcal{F}}_{1,c}$ queries the random oracle $\mathcal{F}(K, x_1||x_2)$ to obtain this value. The process continues until reaching x_n which will create the tuple $(x_{\_} = x_n, y_{\_}, c_{\_}, z_{\_}, t_{\_} = 1)$ where $z_{\_} = \mathcal{F}(K, x_1||x_2||\dots||x_n) = F^{\mathcal{F}}_{1,c}(K, X)$. □

It follows then that this game is a syntactical rewrite of the pervious game and the view of D will not change when it interacts with ($F^{\mathcal{F}}_{1,x}$ or $F^{S_{1,x},R_{1,x}}_{2,x}$) and ($F^{\mathcal{F}}_{1,y}$ or $F^{S_{1,y},R_{1,y}}_{2,y}$) and ($F^{\mathcal{F}}_{1,c}$ or $F^{S_{1,c},R_{1,c}}_{2,c}$), except that we now have to account for queries $q_1 \in \{\{0,1\}^m\}^*$ since these will update the tables $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$. Previously, when calculating the probabilities we only considered queries q_2 and q_3 because they were the only queries that will access and interact with the tables $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$. However, accounting for q_1 is slightly less straightforward than q_2 and q_3 since q_1 have a variable

length. Let L denote the maximum length of q_1 (we assume that L is divisible by m). What we are concerned about here is how many times a single q_1 query accesses and probably updates $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ because this is what affects the probability, and we obtain this number by L/m, where m is the length of a single block (recall that a query q_1 will be partitioned into several m-bit blocks which will then be processed sequentially in order by $F_{2,x}^{S_{1,x},R_{1,x}}, F_{2,y}^{S_{1,y},R_{1,y}}, F_{2,c}^{S_{1,c},R_{1,c}}$). Additionally, in this game $F_{2,x}^{S_{1,x},R_{1,x}}, F_{2,y}^{S_{1,y},R_{1,y}}, F_{2,c}^{S_{1,c},R_{1,c}}$ make extra finalising calls to $R_{1,x}, R_{1,y}, R_{1,c}$, and thus the probability of collisions in these calls need to be accounted for[4], which is implicit with the q_1 queries. Therefore, the final probability of this game is the sum of the failure conditions in $G(3)$ and $G(4)$ given the additional q_1 queries.

$$
\begin{aligned}
P_5^x &= \Pr[D^{F_2^{\mathcal{F}},S_{1,x}^{\mathcal{F}},R_{1,x}^{\mathcal{F}}} = 1] \\
&\leq \left(2^n(3(q_1 \cdot L/m) + 5(q_1 \cdot L/m)^2) + (q_1 \cdot L/m)\right)/2^{2n} + \left((q_1 \cdot L/m)\right)/2^m \\
P_5^y &= \Pr[D^{F_2^{\mathcal{F}},S_{1,y}^{\mathcal{F}},R_{1,y}^{\mathcal{F}}} = 1] \\
&\leq \left(5(q_1 \cdot L/m) + 5(q_1 \cdot L/m)^2\right)/2^n + \left(2^2(q_1 \cdot L/m)^2 + (q_1 \cdot L/m)\right)/2^{m+n} \\
P_5^c &= \Pr[D^{F_2^{\mathcal{F}},S_{1,c}^{\mathcal{F}},R_{1,c}^{\mathcal{F}}} = 1] \\
&\leq \left((q_1 \cdot L/m)(3 \cdot 2^m + 2^{m+1}(q_1 \cdot L/m) + 2^n(q_1 \cdot L/m) + 1)\right)/2^{m+n} \\
&+ \left((q_1 \cdot L/m) + 2(q_1 \cdot L/m)^2\right)/2^n
\end{aligned}
$$

Game 6. In this game we modify simulators $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}, i \in \{x, y, c\}$ to remove their dependency on $\mathcal{F}$ making them self-dependant (they now generate all their responses independently and uniformly at random); the new simulators $S_{2,x}, R_{2,x}, S_{2,x}^{-1}, R_{2,x}^{-1}, S_{2,y}, R_{2,y}, S_{2,y}^{-1}, R_{2,y}^{-1}, S_{2,c}, R_{2,c}, S_{2,c}^{-1}, R_{2,c}^{-1}$ are defined in figures 5-8 and 5-9. Unlike $G(3)$ and $G(4)$, where we modified the S and R simulators separately in different games, we had to modify both simulators simultaneously in this game because they are accessing the same shared table $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ and a single simulator is no longer generating a full tuple for every query it receives. That is, if we only modify one of them, the table will suffer from inconsistencies since then one of the simulators will interact with it differently than the other. Although the new simulators $S_{2,i}, R_{2,i}, S_{2,i}^{-1}, R_{2,i}^{-1}, i \in \{x, y, c\}$ still access the tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, respectively, they no longer check for any failure condition, and they do not need to because they guarantee collision-freeness as we prove below. Also, as stated above, they are not required to generate complete tuples every time they are queried, that is, a query to $S_{2,x}$ or $S_{2,x}^{-1}$ will create the c, y, z fields

[4] A subtle technical issue is when $F_{2,x}^{S_{1,x},R_{1,x}}$ calls $R_{1,x}$ to finalise an n-bit string ($R_{1,x}$ originally handles m-bit strings). To resolve this problem, in section 5.2 we propose padding the n-bits by 0 $m - n$ bits, which will not affect the proof, and have a negligible effect on the running time.

of a tuple (setting $x = \perp$), a query to $S_{2,y}$ or $S_{2,y}^{-1}$ will create x, c, z (setting $y = \perp$), and a query to $S_{2,c}$ or $S_{2,c}^{-1}$ will create x, y, c (setting $z = \perp$), while a query to $R_{2,x}$ or $R_{2,x}^{-1}$ will create x, c (setting $y = z = \perp$), a query to $R_{2,y}$ or $R_{2,y}^{-1}$ will create y, c (setting $x = z = \perp$), and finally a query to $R_{2,c}$ or $R_{2,c}^{-1}$ will create c, z (setting $x = y = \perp$). Thus, unlike $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}$, with $S_{2,i}, R_{2,i}^{-1}, S_{2,i}^{-1}, R_{2,i}^{-1}, i \in \{x, y, c\}$ at least two queries are now required in order to create a new complete tuple in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. Since we do not need to check for sequenced and singular queries, as queries from $F_2^{S_2,R_2}$ will now be already in sequence, and direct queries to $S_{2,i}, R_{2,i}^{-1}, S_{2,i}^{-1}, R_{2,i}^{-1}, i \in \{x, y, c\}$ will automatically be singular (as per their definitions in figures 5-8 and 5-9), simulators $S_{2,i}, R_{2,i}^{-1}, S_{2,i}^{-1}, R_{2,i}^{-1}$ will now drop the field t from $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. The easiest way to calculate the distinguishing probability of this game is to observe the differences between the simulators $S_{2,i}, R_{2,i}, S_{2,i}^{-1}, R_{2,i}^{-1}$ and $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}, i \in \{x, y, c\}$, and demonstrate how changes in the new simulators may affect D's view from $G(5)$ to $G(6)$.

Simulator $S_{2,x} : (\rightarrow, c, y)$
if $(c_i, y_i, z_i) \in \mathcal{T}_x^K \wedge (c, y) = (c_i, y_i)$
  return z_i
else return
  $z \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_x^K(\hat{z}) \cup \mathcal{T}_x^K(\hat{y}) \cup IV\}$

Simulator $S_{2,x}^{-1} : (\leftarrow, c, z)$
if $(c_i, y_i, z_i) \in \mathcal{T}_x^K \wedge (c, z) = (c_i, z_i)$
  return y_i
else return
  $y \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_x^K(\hat{z}) \cup \mathcal{T}_x^K(\hat{y}) \cup IV\}$

Simulator $S_{2,y} : (\rightarrow, x, c)$
if $(x_i, c_i, z_i) \in \mathcal{T}_y^K \wedge (x, c) = (x_i, c_i)$
  return z_i
else return
  $z \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_y^K(\hat{z}) \cup \mathcal{T}_y^K(\hat{y}) \cup IV\}$

Simulator $S_{2,y}^{-1} : (\leftarrow, x, z)$
if $(x_i, c_i, z_i) \in \mathcal{T}_y^K \wedge (x, z) = (x_i, z_i)$
  return c_i
else return
  $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_y^K(\hat{c})\}$

Simulator $S_{2,c} : (\rightarrow, x, y)$
if $(x_i, y_i, c_i) \in \mathcal{T}_c^K \wedge (x_i, y_i) = (x, y)$
  return c_i
else return
  $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_c^K(\hat{c})\}$

Simulator $S_{2,c}^{-1} : (\leftarrow, x, c)$
if $(x_i, y_i, c_i) \in \mathcal{T}_c^K \wedge (x_i, c_i) = (x, c)$,
  return y_i
else return
  $y \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_c^K(\hat{z}) \cup \mathcal{T}_c^K(\hat{y}) \cup IV\}$

Figure 5-8: Simulators $S_{2,x}, S_{2,x}^{-1}, S_{2,y}, S_{2,y}^{-1}, S_{2,c}, S_{2,c}^{-1}$

Basically, there are two main differences between the new simulators $S_{2,i}, R_{2,i}, S_{2,i}^{-1}, R_{2,i}^{-1}$ and the old ones $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}, i \in \{x, y, c\}$:

1. A new single query to the new simulators does not result in creating a full tuple.
2. The c and z values are no longer being generated by a $\mathcal{RO}$, instead they are always being generated uniformly at random.

Simulator $R_{2,x:}(\rightarrow, K, x)$
if $(x_i, c_i) \in \mathcal{T}_x^K \wedge x_i = x$
  return c_i
else return $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_x^K(\hat{c})\}$

Simulator $R_{2,y:}(\rightarrow, K, y)$
if $(y_i, c_i) \in \mathcal{T}_y^K \wedge y_i = y$
  return c_i
else return $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_y^K(\hat{c})\}$

Simulator $R_{2,c} : (\rightarrow, K, c)$
if $(c_i, z_i) \in \mathcal{T}_c^K \wedge c_i = c$
  return z_i
else return
  $z \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_c^K(\hat{z}) \cup \mathcal{T}_c^K(\hat{y}) \cup IV\}$

Simulator $R_{2,x}^{-1} : (\leftarrow, K, c)$
if $(x_i, c_i) \in \mathcal{T}_x^K \wedge c_i = c$, **return** x_i
else return $x \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_x^K(\hat{x})\}$

Simulator $R_{2,y}^{-1} : (\leftarrow, K, c)$
if $(y_i, c_i) \in \mathcal{T}_y^K \wedge c_i = c$
  return y_i
else return
  $y \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_y^K(\hat{z}) \cup \mathcal{T}_y^K(\hat{y}) \cup IV\}$

Simulator $R_{2,c}^{-1} : (\leftarrow, K, z)$
if $(c_i, z_i) \in \mathcal{T}_c^K \wedge z_i = z$
  return c_i
else return $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_c^K(\hat{c})\}$

Figure 5-9: Simulators $R_{2,x}, R_{2,x}^{-1}, R_{2,y}, R_{2,y}^{-1}, R_{2,c}, R_{2,c}^{-1}$

However, even these changes do not affect D's view in this game because they will not cause collisions. Below, we prove that this game, with the modifications introduced to the simulators, is collision-free.

Lemma 5.4.9 (Collision freeness of $G(6)$). *Simulators $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}$ accessed by $F_{2,i}^{S_{2,i},R_{2,i}}$, where $i \in \{x, y, c\}$, guarantee collision, prefix and fixed-point freeness.*

Proof. Collisions between two tuples $(x_p, y_p, c_p, z_p), (x_q, y_q, c_q, z_q), p \neq q$, occur if $y_p = z_q$ or $z_p = z_q$ or $y_p = y_q$, while fixed points occur if $y_i = z_i$ or $z_i = IV$, where $i \in \{p, q\}$. We show that as long as $F_{2,i}^{S_{2,i},R_{2,i}}$ use $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}$, where $i \in \{x, y, c\}$, to process any query they receive, the collision and fixed point scenarios above are impossible. The proof follows from the definitions of $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}$. Any y_p of any tuple may not collide with any y_q of any other tuple or any z_p of the same tuple, or any z_q of any other tuple because y_p is always begin generated uniformly at random as follows: $y_p \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_i^K(\hat{y}) \cup \mathcal{T}_i^K(\hat{z}) \cup IV\}$, which excludes values of all the y and z fields of the tuples already exist in $\mathcal{T}_i^K, i \in \{x, y, c\}$. Similarly, a new z_i value is generated as follows: $z \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_K(\hat{z}) \cup \mathcal{T}_K(\hat{y}) \cup IV\}$, which again excludes all the values of the y and z fields of the tuples already exist in $\mathcal{T}_i^K, i \in \{x, y, c\}$ and thus thwarts any possible collision with them. Therefore, collisions between any y and any z in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ are impossible. This also immediately implies that prefix collisions (where $y = z$) are impossible too. Fixed-point-freeness follows since both y and z are generated excluding IV and other existing values of y and z in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. □

Consequently, since the view of D will only change if collisions were found in the previous game, the probability in $G(6)$ is the probability that the simulators $S^{\mathcal{F}}_{1,i}, R^{\mathcal{F}}_{1,i}$, $(S^{\mathcal{F}}_{1,i})^{-1}, (R^{\mathcal{F}}_{1,i})^{-1}, i \in \{x, y, c\}$ in $G(5)$ will output collisions while the modified simulators $S_{2,i}, S^{-1}_{2,i}, R_{2,i}, R^{-1}_{2,i}$ in $G(6)$ will not, which is 0. Thus,

$$\begin{aligned} P^x_6 = \Pr[D^{F^{S_{2,x},R_{2,x}}_{2,x},S_{2,x},R_{2,x}}] &= P^x_5 \\ P^y_6 = \Pr[D^{F^{S_{2,y},R_{2,y}}_{2,y},S_{2,y},R_{2,y}}] &= P^y_5 \\ P^c_6 = \Pr[D^{F^{S_{2,c},R_{2,c}}_{2,c},S_{2,c},R_{2,c}}] &= P^c_5 \end{aligned}$$

Game 7. In this game we remove the shared tables $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ and modify the simulators $S_{2,i}, S^{-1}_{2,i}, R_{2,i}, R^{-1}_{2,i}, i \in \{x.y, c\}$ to maintain their own separate private tables. The new simulators $S_{3,i}, S^{-1}_{3,i}, R_{3,i}, R^{-1}_{3,i}$ maintain the new tables $\mathcal{T}^K_{i,S}$ (maintained by $S_{2,i}, S^{-1}_{2,i}$), and $\mathcal{T}^K_{i,R}$ (maintained by $R_{2,i}, R^{-1}_{2,i}$), where $i \in \{x, y, c\}$. These new tables contain tuples of the format (c, y, z) for $\mathcal{T}^K_{x,S}, (x, c)$ for $\mathcal{T}^K_{x,R}, (x, c, z)$ for $\mathcal{T}^K_{y,S}, (y, c)$ for $\mathcal{T}^K_{y,R}, (x, y, c)$ for $\mathcal{T}^K_{c,S}, (c, z)$ for $\mathcal{T}^K_{c,R}$. The definitions of the new simulators $S_{3,i}, S^{-1}_{3,i}, R_{3,i}$, $R^{-1}_{3,i}$ are similar to the definitions of the simulators $S_{2,i}, S^{-1}_{2,i}, R_{2,i}, R^{-1}_{2,i}$ in figures 5-8 and 5-9, except that the new simulators now update and refer to their values from their own (unshared) tables. Since the new simulators $S_{3,i}$ have no access to $\mathcal{T}^K_{i,R}$ and $R_{3,i}$ have no access to $\mathcal{T}^K_{i,S}$, one may be inclined to think that $c_p \in \mathcal{T}^K_{i,S}, c_q \in \mathcal{T}^K_{i,R}$ such that $c_p = c_q$ implies a collision. However, this is not the case, the field c here acts as a connecting variable to link the two tables. In fact, $F^{S_{3,i},R_{3,i}}_2$ will always create this linking c among $\mathcal{T}^K_{i,S}$ and $\mathcal{T}^K_{i,R}$ to process its queries.

Lemma 5.4.10 (Collision freeness within query tables). *The tables $\mathcal{T}^K_{i,S}$ and $\mathcal{T}^K_{i,R}$, maintained separately by simulators $S_{3,i}$ and $R_{3,i}$, respectively, where $i \in \{x, y, c\}$, may not exhibit collisions in the common field c.*

Proof. A genuine collision in the c field means either $\mathcal{T}^K_{i,S}$ has $c_p = c_q$ while $p \neq q$ or $\mathcal{T}^K_{i,R}$ has $c_a = c_b$ while $a \neq b$; but, this cannot happen because both $S_{3,i}$ and $R_{3,i}$ generate c excluding all the other c values of the existing tuples in their respective tables, that is, $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}^K_{i,S}(\hat{c})\}$, $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}^K_{i,R}(\hat{c})\}$. Thus, there may at most be one value of c in a table, but that value may exist in both tables which does not imply a collision, rather it acts as a connecting point between the two tables. □

Even though collisions of c may not be encountered, prefix collisions and fixed points are still possible. For example, when $S_{3,y}$ generates z, it has no way to exclude the values of y (to prevent a prefix collision) because there is no y field in $\mathcal{T}^K_{y,S}$ and it has no access to $\mathcal{T}^K_{y,R}$. Similar arguments apply for other simulators (i.e., there is no

single table that contains both y and z fields). Also, a fixed point, where $y = z$, cannot be prevented for the same reason. To simulate practical configurations of the ideal cipher and integration function, we also cannot exclude the IV when generating z or y, making a fixed point such as $y = IV$ or $z = IV$ possible. Therefore, the probability in this game is the probability that the simulators will output either prefix collision, or fixed points. Since the occurrence probability of a prefix collision can be upper bounded by the birthday attack, the overall probability of this game is:

$$\begin{aligned} P_7^x = \Pr[D^{F_{2,x}^{S_{3,x},R_{3,x}},S_{3,x},R_{3,x}}] &\leq \left((q_1 \cdot L/m + q_2 + q_3)^2 + 2(q_1 \cdot L/m + q_2 + q_3)\right)/2^n \\ P_7^y = \Pr[D^{F_{2,y}^{S_{3,y},R_{3,y}},S_{3,y},R_{3,y}}] &\leq \left((q_1 \cdot L/m + q_2 + q_3)^2 + 2(q_1 \cdot L/m + q_2 + q_3)\right)/2^n \\ P_7^c = \Pr[D^{F_{2,c}^{S_{3,c},R_{3,c}},S_{3,c},R_{3,c}}] &\leq \left((q_1 \cdot L/m + q_2 + q_3)^2 + 2(q_1 \cdot L/m + q_2 + q_3)\right)/2^n \end{aligned}$$

where the 2 instances of $(q_1 \cdot L/m + q_2 + q_3)/2^n$ signify the 2 possible fixed points, namely $y = IV$ or $z = IV$.

Game 8. We can now replace $\mathcal{F}_{2,x}^{S_{3,x},R_{3,x}}, \mathcal{F}_{2,y}^{S_{3,y},R_{3,y}}, \mathcal{F}_{2,c}^{S_{3,c},R_{3,c}}$ by $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$ and $S_{3,x}, S_{3,y}, S_{3,c}$ by $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$, and $R_{3,x}, R_{3,y}, R_{3,c}$ by $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$. Clearly, the distinguishing probability of this game is similar to the previous one as the view of D does not change by the replacements as detailed above:

$$\begin{aligned} P_8^x = \Pr[D^{\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x},\mathcal{H}_x,\mathcal{G}_x} = 1] = P_7^x \\ P_8^y = \Pr[D^{\hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y},\mathcal{H}_y,\mathcal{G}_y} = 1] = P_7^y \\ P_8^c = \Pr[D^{\hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c},\mathcal{H}_c,\mathcal{G}_c} = 1] = P_7^c \end{aligned}$$

Finally, we add the distinguishing probabilities calculated throughout the proof and obtain the full distinguishing bound for each construction. The running time of the simulator is (number of queries) $\times$ (largest query), that is $t_S \leq (q_1 \cdot L/m + q_2 + q_3) \cdot (m+n)$ and is similar in all games. This completes the proof. □

5.5 Summary

In this chapter we proved that the iMD constructions, proposed in chapter 4, are indifferentiable from $\mathcal{RO}$ when their underlying primitives are modelled as ideal primitives. We developed a generic proof based on the popular game-playing technique showing that a distinguisher cannot differentiate between systems representing our iMD constructions and another representing a $\mathcal{RO}$.

Chapter 6

Indistinguishability and Unforgeability of the iMD Constructions

In this chapter, we conclude the contributions of the thesis by discussing the indistinguishability and unforgeability of the integrated-key hash functions. We first show that hash functions constructed in the integrated-key setting are indistinguishable from their variants in the conventional dedicated-key setting which (using the result from chapter 5) implies that they are also immediately indistinguishable from Pseudorandom Functions (PRF). We then use this result to further show that the previously proposed x-iMD, y-iMD, c-iMD constructions are unforgeable when used as Message Authentication Code (MAC), which is a central hash functions application.

6.1 Introduction

In chapters 4 and 5, we introduced the integrated-key setting where a keyless hash function is transparently transformed to a keyed hash function, creating a family of functions indexed by different keys. We then proposed several integrated-key constructions (called the iMD constructions) and proceeded to prove that they preserve various important hash function security properties, from collision resistance, pre-image resistance and 2nd pre-image resistance to being indifferentiable from $\mathcal{RO}$. In this chapter, we conclude the security analysis of the iMD constructions by providing discussions and proofs on their indistinguishability and unforgeability (these proofs may also be applicable to other integrated-key hash functions exhibiting similar structures to the iMD

constructions). Although the results are logically separate, in the unforgeability proof we used the indistinguishability result to model the composition of the compression and integration functions as a single entity.

Indistinguishability. In the first part of this chapter we show that the iMD constructions are indistinguishable from their variants in the dedicated-key setting. That is, if we visualise the compression functions and integration functions of the iMD constructions as single black-box entities, accepting a message block and a key input, then we basically obtain the iMD variants in the dedicated-key setting. Obviously, since the only difference between the iMD constructions is where each construction places its integration function around the compression function, this abstraction makes x-iMD, y-iMD, c-iMD identical. To argue about the indistinguishability between the iMD constructions and their dedicated-key variant, we use a *collision-based* proof approach that adopts the indistinguishability framework due to Maurer [107] (this framework is based on the notion of random systems, which we generalise in section 6.2):

Definition 6.1.1 (Indistinguishability). *The random systems $\boldsymbol{F}$ and $\boldsymbol{R}$ are said to be (computationally) indistinguishable if for any distinguisher D with access to both $\boldsymbol{F}$ and $\boldsymbol{R}$ (but does not know which system is which), the distinguishing advantage:*

$$\left|\Pr[D^{\boldsymbol{F}_k} \rightarrow 1] - \Pr[D^{\boldsymbol{R}_k} \rightarrow 1]\right|$$

is negligible in the security parameter k.

In definition 6.1.1, the security parameter k is basically the number of queries the distinguisher D sends to the systems before deciding to output either 1 or 0. Outputting 1 means that D believes that it is interacting with a particular system and outputting 0 means that D believes it is interacting with the other system. While it is not necessarily always the case, D is usually programmed to output 1 when it thinks it is interacting with the real system (the system that needs to be proven indistinguishable from another ideal system) and outputs 0 to indicate that it thinks it is interacting with the ideal system. For D to succeed, it should be able to make many correct guesses, this will maximise the distinguishing advantage as per definition 6.1.1. In particular, we aim to prove that a distinguisher D cannot distinguish between two, identically-packaged, keyed hash function C^{f_K} and $\hat{C}^{f,g_K}$, one with access to a keyed compression function f_K (representing a dedicated-key hash function), while the other with access to a keyless compression function f and a keyed integration function g_K (representing an integrated-key hash function); we denote such indistinguishability by $C^{f_k} \bowtie \hat{C}^{f,g_k}$. This naturally implies $f_K \bowtie (f \star g_k)$, where $\star$ is some composition operation.

Indistinguishability here is the right notion because the adversary is given a black-box access to both systems and it should not be able to distinguish between them. Unlike the indifferentiability framework [108], the indistinguishability framework does not allow the adversary to access the internal components of a system and here we aim to show that if we model the composition of the compression function and integration function as a single component, no adversary will be able to (computationally) distinguish it from a single component with the same domain and range (i.e., a dedicated-key compression function; one that naturally accepts a key).

Message Authentication Code. As discussed in section 2.5, MACs (Message Authentication Code) are cryptographic primitives used to preserve data integrity and authenticity. Basically, a MAC algorithm accepts a message M and a (secret) key K and produces a tag T, that is, $\text{MAC}(K, M) = T$. Once T is produced, a sender transmits the message-tag pair (M, T) to a receiver which, in turn, verifies the integrity of the message M using a verification algorithm. The verification algorithm accepts the received message-tag pair (M', T') and outputs 1 if $\text{MAC}(K, M') = T'$, in which case the receiver accepts M', otherwise it output 0, in which case the receiver rejects M'. MACs preserve integrity because if M (the original message) was tampered with in transit, $\text{MAC}(K, M) \neq T$ with high probability. MACs also preserve authenticity because only parties who have access to the secret key K can generate a valid message-tag pair. One of the most common approaches for designing MACs is based on hash functions, the prime example is HMAC [21]. In this approach, a secretly keyed[1] hash function $F_K : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$ is used to produce the tag.

MAC schemes need to be *unforgeable*, which means that no adversary A can forge a message-tag pair (M', T') such that $\text{MAC}(K, M') = T'$, without access to the secret key K. In particular, A is given access to a MAC scheme which it can query by sending messages and receive the corresponding tags. After a number of such queries, A outputs a message M' that was never queried to the MAC scheme, and claims that a particular T' is the tag of M'. Formally, the advantage (success probability) of an adversary A against a MAC scheme built from a keyed hash function F_K (the probability that A will succeed in forging against F_K) is as follows:

$$\mathbf{Adv}^{\text{mac}}_{F_K}(\text{A}) = \Pr\left[K \xleftarrow{\$} \mathcal{K}, (M, T) \xleftarrow{\$} \text{A}^{F_K} : F_K(M) = T \wedge M \text{ not queried}\right]$$

According to the definition, an adversary A succeeds in the unforgeability game as long

[1] In MAC, the hash function has to be keyed and the key should remain secret. This, however, does not preclude keyless hash functions from being used as MACs as there are ways to key them, such as using the IV as a key or appending the key to the message [81], though these may be forgeable [48].

as it manages to create a message-tag pair that will pass the verification process, in this case the scheme is said to be forgeable, even if the adversarially created message-tag pair is completely different than the one originally sent by a genuine party. Note that here a sender transmits the message M in plaintext because MAC only aims to test whether M has been somehow altered (maliciously or inadvertently) in transit.

To prove the unforgeability of a hash-based MAC scheme, some existing analysis, e.g., [22, 19, 159], made unnecessarily strong assumptions on the compression function modelling it as a Pseudorandom Function (PRF) [76]. However, since in cryptography it is always desirable to carry out proofs based on as weak assumptions as possible, we will prove the unforgeability of our constructions based on the sole assumption that the underlying primitive is a good MAC, which is strictly weaker assumption than PRF. Bellare and An [7] followed this approach while proving the unforgeability of their Nested Iteration (NI) mode. Similarly, Maurer and Sjödin introduced a general unforgeability proof approach and used it to prove that their Chain Shift (CS) and Rotate Shift (RS) constructions [109] are unforgeable.

Chapter Outline. This chapter is organised as follows. In section 6.2, we prove that a hash function constructed in the integrated-key setting is (computationally) *indistinguishable* from its variant in the dedicated-key setting, which implies that they are also indistinguishable from Pseudorandom Functions (PRF). Using this indistinguishability result, we further prove in section 6.3 that the iMD constructions are unforgeable.

6.2 Composition and Indistinguishability

Essentially, in an integrated-key hash function, both the compression function f and the integration function g can be visualised as a single composite entity $\mathbf{R}$ that, when treated as a black box, is a dedicated-key compression function (a compression function that admits a key). Thus, intuitively, proofs for $f \star g$ can naturally be reduced to $\mathbf{R}$, where $\star$ is some composition operation. In this chapter, we formalise this argument. To provide concrete security analysis, we model $\mathbf{R}$ as a *generalised random system* (GRS), which is a generalisation of Maurer's random systems notion introduced in [107].

Definition 6.2.1 (Generalised Random System (GRS)). *A Generalised Random System $(\mathcal{X}_1 \dots \mathcal{X}_n \rightarrow \mathcal{Y}_1 \dots \mathcal{Y}_m)$-$\boldsymbol{R}$ is a transformation that given an input $U_i = \{X_1^{U_i} \in \mathcal{X}_1, \dots, X_n^{U_i} \in \mathcal{X}_n\}$, returns a conditionally probabilistic output $V_i = \{Y_1^{V_i} \in \mathcal{Y}_1, \dots, Y_m^{V_i} \in \mathcal{Y}_m\}$. $\boldsymbol{R}$ is stateful if $\boldsymbol{R}(U_i) = P^{\boldsymbol{R}}_{V_i|U^iV^{i-1}}$ (future outputs depend on previous input/outputs) and stateless if $\boldsymbol{R}(U_i) = P^{\boldsymbol{R}}_{V_i|U_i}$ (future outputs depend on only the current inputs) for $i \geq 1$, where $U^i = U_1, \dots, U_i$ and $V^{i-1} = V_1, \dots, V_{i-1}$. A GRS may either*

be atomic (cannot be decomposed to sub GRS') or composite (formed by the composition of several atomic or composite GRS' and can be decomposed).

Compared to Maurer's notion of random systems, the GRS notion covers a wider range of applications where cryptosystems accept several input parameters and return more than one output. As stated in definition 6.2.1, GRS' are characterised by their type and input/output spaces. In this chapter, we will always attribute the GRS' to single output space $\mathcal{Y}$ and either single $\mathcal{X}$ or double $(\mathcal{X}, \mathcal{K})$ input spaces, while using both atomic or composite GRS' as necessary. We use the notation $(\mathcal{X}, \mathcal{Y})$-$\mathbf{R}$ to denote an atomic GRS $\mathbf{R}$, and the notation $(\mathcal{X}, \mathcal{Y})$-$\mathbf{S}^{\mathbf{F}\star\mathbf{G}}$ to denote a composite GRS $\mathbf{S}$ formed by the composition of $\mathbf{F}$ and $\mathbf{G}$ sub GRS' ($\mathbf{F}$ and $\mathbf{G}$ can themselves be either atomic or composite). Composition of GRS' can be parallel or sequential, but in this chapter we only consider sequential composition, which can further be full or partial.

Definition 6.2.2 (Full sequential composition). *A GRS $(\mathcal{X}_1 \dots \mathcal{X}_n \rightarrow \mathcal{Y}_1 \dots \mathcal{Y}_m)$-$\boldsymbol{R}$ is a full sequential composite GRS if it is formed by the composition of k sub GRS' $(\mathcal{X}_1 \dots \mathcal{X}_n, \mathcal{Z}_1 \dots \mathcal{Z}_p)$-$\boldsymbol{F}_1$, $(\mathcal{Z}_1 \dots \mathcal{Z}_p, \mathcal{V}_1 \dots \mathcal{V}_q)$-$\boldsymbol{F}_2, \dots, (\mathcal{G}_1 \dots \mathcal{G}_r, \mathcal{Y}_1 \dots \mathcal{Y}_m)$-$\boldsymbol{F}_k$, where $k > 1$, denoted by $\boldsymbol{R}^{\boldsymbol{F}_1 \circ \boldsymbol{F}_2 \circ \cdots \circ \boldsymbol{F}_k}$, such that $\boldsymbol{R}^{\boldsymbol{F}_1 \circ \boldsymbol{F}_2 \cdots \circ \boldsymbol{F}_k}(U_i) = \boldsymbol{F}_k(\dots \boldsymbol{F}_2(\boldsymbol{F}_1(U_i)) \dots)$, where $U_i = \{X_1^{U_i} \in \mathcal{X}_1, \dots, X_n^{U_i} \in \mathcal{X}_n\}$. Generally, $\boldsymbol{R}^{\boldsymbol{F} \circ \boldsymbol{G}} \neq \boldsymbol{R}^{\boldsymbol{G} \circ \boldsymbol{F}}$.*

Definition 6.2.3 (Partial sequential composition). *A GRS $(\mathcal{X}_1 \dots \mathcal{X}_p \rightarrow \mathcal{Y}_1 \dots \mathcal{Y}_q)$-$\boldsymbol{P}$ is a partial sequential composite of GRS' $(\mathcal{Z}_1 \dots \mathcal{Z}_n, \mathcal{V}_1 \dots \mathcal{V}_m)$-$\boldsymbol{G}$, and $(\mathcal{X}_1 \dots \mathcal{X}_p, \mathcal{Y}_1 \dots \mathcal{Y}_q)$-$\boldsymbol{F}$, denoted $\boldsymbol{P}_{z_c}^{\boldsymbol{G} \triangle \boldsymbol{F}}$, where $z \in \{R, L\}$, $|\mathcal{X}_1 \dots \mathcal{X}_p| = a$, $|\mathcal{Y}_1 \dots \mathcal{Y}_q| = b$, $|\mathcal{Z}_1 \dots \mathcal{Z}_n| = c$, $|\mathcal{V}_1 \dots \mathcal{V}_m| = d$, and $a > c$, if it is formed by composing $\boldsymbol{F}$ and $\boldsymbol{G}$ as follows:*

$$\boldsymbol{P}_{z_c}^{\boldsymbol{G} \triangle \boldsymbol{F}} = \begin{cases} \boldsymbol{F}(\boldsymbol{G}([U_i]^c) || [U_i]_{a-c}) & \text{if } z = R, \\ \boldsymbol{F}([U_i]^{a-c} || \boldsymbol{G}([U_i]_c)) & \text{if } z = L. \end{cases}$$

where $|U_i| = c$-bits, $U_i = \{X_1^{U_i} \in \mathcal{X}_1, \dots, X_n^{U_i} \in \mathcal{X}_n\}$ and $[U_i]^c$ (respectively, $[U_i]_c$) denotes the most (respectively, least) significant c-bits of U_i. If $z = R$, the composition is called right composition, otherwise it is left composition. Similarly, $\boldsymbol{P}^{\boldsymbol{F} \triangle \boldsymbol{G}}$ is formed by composing $\boldsymbol{F}$ with part of $\boldsymbol{G}$. Generally, $\boldsymbol{P}^{\boldsymbol{F} \triangle \boldsymbol{G}} \neq \boldsymbol{P}^{\boldsymbol{G} \triangle \boldsymbol{F}}$.

6.2.1 Indistinguishability from the Dedicated-key Setting

Using Maurer's indistinguishability framework [107], we prove that the following systems are indistinguishable: (1) $\mathbf{R}$ and $\mathbf{A}^{\mathbf{F} \circ \mathbf{G}}$, (2) $\mathbf{R}$ and $\mathbf{B}_{R_m}^{\mathbf{G} \triangle \mathbf{F}}$, (3) $\mathbf{R}$ and $\mathbf{C}_{L_n}^{\mathbf{G} \triangle \mathbf{F}}$, where $\mathbf{F}$ represents a compression function and $\mathbf{G}$ represents an integration function. The systems $\mathbf{A}^{\mathbf{F} \circ \mathbf{G}}, \mathbf{B}_{R_m}^{\mathbf{G} \triangle \mathbf{F}}, \mathbf{C}_{L_n}^{\mathbf{G} \triangle \mathbf{F}}$ represent the composite systems formed by compos-

ing the compression function and integration function in the c-iMD, x-iMD y-iMD constructions, respectively, while $\mathbf{R}$ represents a dedicated-key compression function.

Theorem 6.2.4 (Integrated-key Dedicated-key Indistinguishability). *The atomic GRS $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\boldsymbol{R}$ is $(t_{\boldsymbol{A}}, q_{\boldsymbol{A}}, \epsilon_{\boldsymbol{A}})$-indistinguishable from the composite GRS $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\boldsymbol{A}^{\boldsymbol{F}_1 \circ \boldsymbol{G}_1}$, $(t_{\boldsymbol{B}}, q_{\boldsymbol{B}}, \epsilon_{\boldsymbol{B}})$-indistinguishable from the composite GRS $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\boldsymbol{B}_{R_m}^{\boldsymbol{G}_2 \vartriangle \boldsymbol{F}_2}$, $(t_{\boldsymbol{C}}, q_{\boldsymbol{C}}, \epsilon_{\boldsymbol{C}})$-indistinguishable from the composite GRS $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\boldsymbol{C}_{L_n}^{\boldsymbol{G}_3 \vartriangle \boldsymbol{F}_3}$, where:*

1. $\epsilon_{\boldsymbol{A}} \leq q_{\boldsymbol{A}}^2/2^n, \quad$ *and* $\quad t_{\boldsymbol{A}} \leq q_{\boldsymbol{A}}(\tau_{\boldsymbol{F}_1} + \tau_{\boldsymbol{G}_1})$
2. $\epsilon_{\boldsymbol{B}} \leq q_{\boldsymbol{B}}^2/2^m, \quad$ *and* $\quad t_{\boldsymbol{B}} \leq q_{\boldsymbol{B}}(c_{mn} \cdot \tau_{\boldsymbol{F}_2} + \tau_{\boldsymbol{G}_2})$
3. $\epsilon_{\boldsymbol{C}} \leq q_{\boldsymbol{C}}^2/2^n, \quad$ *and* $\quad t_{\boldsymbol{C}} \leq q_{\boldsymbol{C}}(c_{mn} \cdot \tau_{\boldsymbol{F}_3} + \tau_{\boldsymbol{G}_3})$

given a distinguisher D sending any number of queries $q_{\boldsymbol{A}}, q_{\boldsymbol{B}}, q_{\boldsymbol{C}}$ of the form (X, K) to the systems and receiving responses Y, where $X \in \mathcal{X}$, $K \in \mathcal{K}$, $Y \in \mathcal{Y}$, and $|X| = m+n$, $|K| = k$, $|Y| = n$. The GRS $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\boldsymbol{A}^{\boldsymbol{F}_1 \circ \boldsymbol{G}_1}$ is a full sequential composite system formed by the composition of $(\mathcal{X} \rightarrow \mathcal{Y})$-$\boldsymbol{F}_1$ and $(\mathcal{Y}, \mathcal{K} \rightarrow \mathcal{Y})$-$\boldsymbol{G}_1$, while GRS' $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\boldsymbol{B}_{R_m}^{\boldsymbol{G}_2 \vartriangle \boldsymbol{F}_2}$ and $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\boldsymbol{C}_{L_n}^{\boldsymbol{G}_3 \vartriangle \boldsymbol{F}_3}$ are partial sequential composite systems, formed by composing $(\mathcal{X}^m, \mathcal{K} \rightarrow \mathcal{X}^m)$-$\boldsymbol{G}_2, (\mathcal{X}^m || \mathcal{X}^n \rightarrow \mathcal{Y})$-$\boldsymbol{F}_2$, and $(\mathcal{X}^n, \mathcal{K} \rightarrow \mathcal{X}^n)$-$\boldsymbol{G}_2, (\mathcal{X}^m || \mathcal{X}^n \rightarrow \mathcal{Y})$-$\boldsymbol{F}_3$, respectively. The parameters $\tau_{\boldsymbol{R}}, \tau_{\boldsymbol{F}}, \tau_{\boldsymbol{G}_1}, \tau_{\boldsymbol{G}_2}, \tau_{\boldsymbol{G}_3}$ are the costs of calling $\boldsymbol{R}$, ($\boldsymbol{F}_1$, $\boldsymbol{F}_2$, $\boldsymbol{F}_3$), $\boldsymbol{G}_1$, $\boldsymbol{G}_2$, $\boldsymbol{G}_3$, respectively, and c_{mn} is a small constant representing the delay caused by partitioning $X^{m+n} = (X^m || X^n)$.

Proof. Let D be a distinguisher with access to two systems, and whose aim is to distinguish between them. D sends queries of the form $(X, K) \in (\mathcal{X}, \mathcal{K})$ and receives responses of the form $Y \in \mathcal{Y}$, where $|X| = m + n$ bits, $|K| = k$-bits, and $|Y| = n$-bits. Also, let $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\mathbf{R}$ be an atomic GRS.

For (1), let $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}$ be a composite GRS formed by the full sequential composition of the atomic GRS' $(\mathcal{X} \rightarrow \mathcal{Y})$-$\mathbf{F}_1$ and $(\mathcal{Y}, \mathcal{K} \rightarrow \mathcal{Y})$-$\mathbf{G}_1$. For (2) and (3), let $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\mathbf{B}_{R_m}^{\mathbf{G}_2 \vartriangle \mathbf{F}_2}$ and $(\mathcal{X}, \mathcal{K} \rightarrow \mathcal{Y})$-$\mathbf{C}_{L_n}^{\mathbf{G}_3 \vartriangle \mathbf{F}_3}$ be composite GRS' formed by the partial composition of the atomic GRS' $(\mathcal{X}^m, \mathcal{K} \rightarrow \mathcal{X}^m)$-$\mathbf{G}_2, (\mathcal{X}^{m+n} \rightarrow \mathcal{Y})$-$\mathbf{F}_2$, and $(\mathcal{X}^n, \mathcal{K} \rightarrow \mathcal{X}^n)$-$\mathbf{G}_3, (\mathcal{X}^{m+n} \rightarrow \mathcal{Y})$-$\mathbf{F}_3$, respectively, where $|X \in \mathcal{X}| > |Y \in \mathcal{Y}|$ and $|\mathcal{X}| = m + n$. The 3 distinguishing instances of D, defined in figure 6-1, succeed in their respective games if they detect differences in how $\mathbf{R}$ and $\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}, \mathbf{B}_{R_m}^{\mathbf{G}_2 \vartriangle \mathbf{F}_2}, \mathbf{C}_{L_n}^{\mathbf{G}_3 \vartriangle \mathbf{F}_3}$ behave when responding to their queries. We use the notation $D_{X \bowtie Y}$ to refer to a distinguisher D trying to distinguish between the two systems X and Y; thus, we define three distinguishers: $D_{R \bowtie A}, D_{R \bowtie B}, D_{R \bowtie C}$, see figure 6-1.

First, D fixes a key $K \in \mathcal{K}$ and sends random queries $(X_1, K), (X_2, K), \ldots (X_q, K)$ to $\mathbf{R}$, $\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}$, $\mathbf{B}_{R_m}^{\mathbf{G}_2 \vartriangle \mathbf{F}_2}$, $\mathbf{C}_{L_n}^{\mathbf{G}_3 \vartriangle \mathbf{F}_3}$, where $X_i \in \mathcal{X}$. Figure 6-2 depicts the internal view of

Distinguisher $D_{R\triangleright\!\triangleleft A}$:
$K \xleftarrow{\$} \mathcal{K}$
for $i = 1$ **to** $q_{\mathbf{A}}$ **do**
$X_i \xleftarrow{\$} \mathcal{X}\backslash X^{i-1}$
$Y_i^{\mathrm{R}} \leftarrow \mathrm{R}(X_i, K)$
$Y_i^{\mathrm{A}} \leftarrow \mathrm{A}^{\mathrm{F}_1\circ\mathrm{G}_1}(X_i, K)$
end for
if $Y_i^{\mathrm{R}} = Y_j^{\mathrm{R}} \wedge$
$i \neq j, 1 \leq i, j \leq q_{\mathbf{A}}$
then $D^R \rightarrow 1$
else if $Y_i^{\mathrm{A}} = Y_j^{\mathrm{A}} \wedge$
$i \neq j, 1 \leq i, j \leq q_{\mathbf{A}}$
then $D^A \rightarrow 1$

Distinguisher $D_{R\triangleright\!\triangleleft B}$:
$K \xleftarrow{\$} \mathcal{K}$
for $i = 1$ **to** $q_{\mathbf{B}}$ **do**
$X_i \xleftarrow{\$} \mathcal{X}\backslash X^{i-1}$
$Y_i^{\mathrm{R}} \leftarrow \mathrm{R}(X_i, K)$
$Y_i^{\mathrm{B}} \leftarrow \mathrm{B}^{\mathrm{G}_2\triangle\mathrm{F}_2}(X_i, K)$
end for
if $Y_i^{\mathrm{R}} = Y_j^{\mathrm{R}} \wedge$
$i \neq j, 1 \leq i, j \leq q_{\mathbf{B}}$
then $D^R \rightarrow 1$
else if $Y_i^{\mathrm{B}} = Y_j^{\mathrm{B}} \wedge$
$i \neq j, 1 \leq i, j \leq q_{\mathbf{B}}$
then $D^B \rightarrow 1$

Distinguisher $D_{R\triangleright\!\triangleleft C}$:
$K \xleftarrow{\$} \mathcal{K}$
for $i = 1$ **to** $q_{\mathbf{C}}$ **do**
$X_i \xleftarrow{\$} \mathcal{X}\backslash X^{i-1}$
$Y_i^{\mathrm{R}} \leftarrow \mathrm{R}(X_i, K)$
$Y_i^{\mathrm{C}} \leftarrow \mathrm{C}^{\mathrm{G}_3\triangle\mathrm{F}_3}(X_i, K)$
end for
if $Y_i^{\mathrm{R}} = Y_j^{\mathrm{R}} \wedge$
$i \neq j, 1 \leq i, j \leq q_{\mathbf{C}}$
then $D^R \rightarrow 1$
else if $Y_i^{\mathrm{C}} = Y_j^{\mathrm{C}} \wedge$
$i \neq j, 1 \leq i, j \leq q_{\mathbf{C}}$
then $D^C \rightarrow 1$

Figure 6-1: Distinguishers $D_{R\triangleright\!\triangleleft A}, D_{R\triangleright\!\triangleleft B}, D_{R\triangleright\!\triangleleft C}$

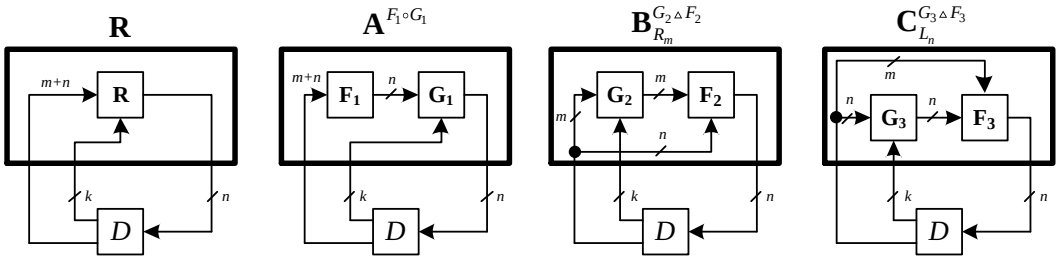


Figure 6-2: Internal structures of systems R, $\mathrm{A}^{\mathrm{F}_1\circ\mathrm{G}_1}$, $\mathrm{B}^{\mathrm{G}_2\triangle\mathrm{F}_2}$, $\mathrm{C}^{\mathrm{G}_3\triangle\mathrm{F}_3}$

systems $\mathbf{R}$, $\mathbf{A}^{\mathbf{F}_1\circ\mathbf{G}_1}$, $\mathbf{B}^{\mathbf{G}_2\triangle\mathbf{F}_2}_{R_m}$, $\mathbf{C}^{\mathbf{G}_3\triangle\mathbf{F}_3}_{L_n}$ and illustrates how they process queries received from D. We model D as a non-adaptive distinguisher such that D generates queries it intends to send to the systems before starting the game[2]. For (1) we let D first interact with $\mathbf{R}$ and outputs 1 every time it finds a collision (i.e., there are X_i and X_j such that $\mathbf{R}(X_i, K) = \mathbf{R}(X_j, K)$ while $X_i \neq X_j$ and $i > j$, where $i, j \in \{1, 2, \ldots, q_{\mathbf{A}}\}$), then D interacts with $\mathbf{A}^{\mathbf{F}_1\circ\mathbf{G}_1}$ and outputs 1 whenever it finds a collision. If the number of collisions in $\mathbf{R}$ is similar to the number of collisions in $\mathbf{A}^{\mathbf{F}_1\circ\mathbf{G}_1}$, the advantage of D is minimised, otherwise it implies that either $\mathbf{R}$ outputs more collisions than $\mathbf{A}^{\mathbf{F}_1\circ\mathbf{G}_1}$ or $\mathbf{A}^{\mathbf{F}_1\circ\mathbf{G}_1}$ outputs more collisions than $\mathbf{R}$, both cases increase the distinguishing advantage of $D_{R\triangleright\!\triangleleft A}$. That is, in conventional indistinguishability proofs, it is usually the case that some real system is argued indistinguishable from another idealised one, but in our case, the GRS $\mathbf{R}$ (representing a keyed compression function) is not ideal and may generate collisions, so one way to argue about its indistinguishability is to adopt this

[2]Compare non-adaptive distinguishers with the adaptive ones where the latter generate its queries in real time during the game such that later queries are generated based on responses of earlier queries.

collision-based approach, where the number of collisions occurring in one system should be about the same number of collisions occurring in another for the two systems to be deemed indistinguishable. In the $\mathbf{R}$ system, collisions depend solely on the structure of the GRS $\mathbf{R}$ since $\mathbf{R}$ is an atomic GRS, but this is clearly not the case with the $\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}$ system as it is a full sequential composition of $\mathbf{F}_1$ and $\mathbf{G}_1$. Thus, collisions in $\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}$ may be the result of (i) collisions in $\mathbf{F}_1$, or (ii) collisions in $\mathbf{G}_1$.

$$\begin{aligned}
\mathbf{Adv}^{\bowtie}_{\mathbf{R},\mathbf{A}}(D) &= \left|\Pr[D^{\mathbf{R}} \rightarrow 1] - \Pr[D^{\mathbf{A}} \rightarrow 1]\right| \\
&= \left|\Pr[\mathrm{Coll}_{\mathbf{R}}] - \Pr[\mathrm{Coll}_{\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}}]\right| \\
&= \left|\Pr[\mathrm{Coll}_{\mathbf{R}}] - (\Pr[\mathrm{Coll}_{\mathbf{F}_1}] + \Pr[\mathrm{Coll}_{\mathbf{G}_1}])\right| \\
&\leq \left|(q^2/2^n) - \left((q^2/2^n) + (q^2/2^n)\right)\right| \leq q^2/2^n
\end{aligned}$$

In $\mathbf{B}^{\mathbf{G}_2 \triangle \mathbf{F}_2}_{R_m}$ and $\mathbf{C}^{\mathbf{G}_3 \triangle \mathbf{F}_3}_{L_n}$ the orientation of the composition is slightly different than $\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}$. In the partial sequential GRS $\mathbf{B}^{\mathbf{G}_2 \triangle \mathbf{F}_2}_{R_m}$, the rightmost m-bit part of the input $X^{m+n} = X^m || X^n$ is first processed by $\mathbf{G}_2$ which will then return an intermediate m-bit string to be concatenated with (the untouched) X^n part of the input and then the whole string is processed by $\mathbf{F}_2$. Like $\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}$, collisions in $\mathbf{B}^{\mathbf{G}_2 \triangle \mathbf{F}_2}$ may be the result of either (i) collisions in $\mathbf{G}_2$, or (ii) collisions in $\mathbf{F}_2$.

$$\begin{aligned}
\mathbf{Adv}^{\bowtie}_{\mathbf{R},\mathbf{B}}(D) &= \left|\Pr[D^{\mathbf{R}} \rightarrow 1] - \Pr[D^{\mathbf{B}} \rightarrow 1]\right| \\
&= \left|\Pr[\mathrm{Coll}_{\mathbf{R}}] - \Pr[\mathrm{Coll}_{\mathbf{B}^{\mathbf{G}_2 \triangle \mathbf{F}_2}}]\right| \\
&= \left|\Pr[\mathrm{Coll}_{\mathbf{R}}] - (\Pr[\mathrm{Coll}_{\mathbf{G}_2}] + \Pr[\mathrm{Coll}_{\mathbf{F}_2}])\right| \\
&\leq \left|(q^2/2^n) - \left((q^2/2^m) + (q^2/2^n)\right)\right| \leq q^2/2^m
\end{aligned}$$

The distinguishing advantage of $D_{R \bowtie C}$ is similar. The running time of $D_{R \bowtie A}$ is q (the number of queries D makes) multiplied by the cost of calling $\mathbf{F}_1$ and $\mathbf{G}_1$. The running times of $D_{R \bowtie B}$ and $D_{R \bowtie C}$ are calculated similarly, except that in this case the input is first partitioned to $X^m || X^n$, hence we include a small constant $c_{m,n}$ to account for the cost of partitioning the input in $D_{R \bowtie B}$ and $D_{R \bowtie C}$, respectively. □

It now follows that as long as $\mathbf{R}$ and $\mathbf{Z}$ are indistinguishable, where $\mathbf{Z} \in \{\mathbf{A}^{\mathbf{F}_1 \circ \mathbf{G}_1}, \mathbf{B}^{\mathbf{G}_2 \triangle \mathbf{F}_2}_{R_m}, \mathbf{C}^{\mathbf{G}_3 \triangle \mathbf{F}_3}_{L_n}\}$, then any cryptosystem (i.e., hash function) $H^{\mathbf{Z}}$, with access to $\mathbf{Z}$, is at least as secure as the cryptosystem $H^{\mathbf{R}}$, with access to $\mathbf{R}$. This is immediate from the result in [108]. Lemma 6.2.5 states this explicitly (using slightly different notation).

Lemma 6.2.5. *Any cryptosystem $H(\boldsymbol{P})$ using $\boldsymbol{P}$ as a component is at least as secure as the cryptosystem $H(\boldsymbol{R})$ obtained from $H(\boldsymbol{P})$ by replacing $\boldsymbol{P}$ with $\boldsymbol{R}$, if and only if $\boldsymbol{R}$ and $\boldsymbol{P}$ are indistinguishable.*

6.2.2 Indistinguishability from PRF

A hash function is PRF-Pr (Pseudorandom Function Preserving) if there is no adversary able to distinguish it from a random function, where the latter is a function that has been chosen randomly based on given domain and range (this does not imply that the output of the function should be random). That is, given a family of hash functions H_K, an adversary with a black-box access to H_K should not be able to distinguish a randomly chosen member function of H_K from a genuinely random function. Succinctly,

$$\mathbf{Adv}^{\mathrm{prf}}_{H_K}(\mathrm{A}) = \left|\Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathrm{A}^{H_K} \rightarrow 1\right] - \Pr\left[R \xleftarrow{\$} Func(Dom, Rng) : \mathrm{A}^{R} \rightarrow 1\right]\right|$$

where R is a randomly chosen function from the set of all functions with domain Dom and range Rng.

In [161] it was shown that if a dedicated-key hash function is PRO-Pr (indifferentiable from Pseudorandom Oracle), then it is trivially PRF-Pr. Since in section 6.2.1 the iMD constructions were shown to be indistinguishable from their variants in the dedicated-key setting, and in chapter 5 it was shown that they are indifferentiable from $\mathcal{RO}$, then this immediately implies that the iMD constructions are also indistinguishable from a PRF (i.e., that they are PRF-Pr). Precisely, in 6.2.1 it was shown that the composition of a keyless compression function $f : \{0,1\}^{m+n} \rightarrow \{0,1\}^n$ and a keyed integration function, either $g_1 : \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m$ or $g_2 : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$, is indistinguishable from a dedicated-key compression function $f_K : \{0,1\}^k \times \{0,1\}^{m+n} \rightarrow \{0,1\}^n$, which implies that the constructions x-iMD^{f_x,g_1,g_2}, y-iMD^{f_y,g_2}, c-iMD^{f_c,g_2}, with access to f_x, f_y, f_c, g_1, g_2, are actually dedicated-key function as long as the compositions $f_x \star g_1, f_y \star g_2, f_c \star g_2$ are accessed in a black-box manner, that is x-iMD^{f_x,g_1,g_2}, y-iMD^{f_y,g_2}, c-iMD^{f_c,g_2}, are indistinguishable from iMD^{f_K}. Additionally, if we remove the last call to g_2 in the iMD^{f_K} constructions, they become the plain Merkle-Damård in the dedicated-key setting, which was shown to be PRF-Pr in [25].

6.3 Unforgeability of the iMD Constructions

When proving the unforgeability of a construction, the underlying compression function needs to be keyed appropriately. Thus, in our iMD constructions, we cannot give direct access for an adversary A to the keyless compression functions f_x, f_y, f_c of the x-iMD, y-iMD, c-iMD constructions. Instead, in each construction we treat the keyless compression function and the keyed integration function as a single composite component for which we can give A oracle access to (indeed, we can do that as per our indistin-

guishability result in section 6.2.1). For the rest of the chapter, the iMD constructions have black-box access to $h_x, h_y, h_c : \{0,1\}^k \times \{0,1\}^{m+n} \to \{0,1\}^n$, and are denoted by $x\text{-iMD}^{h_x,g_2}, y\text{-iMD}^{h_y,g_2}, c\text{-iMD}^{h_c,g_2}$, where $h_x = g_1 \triangle f_x, h_y = g_2 \triangle f_y, h_c = f_c \circ g_2$ and $g_1 : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^m, g_2 : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$.

Theorem 6.3.1 (Unforgeability of x-iMD, y-iMD, c-iMD). *Let the keyed composite functions $h_x, h_y, h_c : \{0,1\}^k \times \{0,1\}^{m+n} \to \{0,1\}^n$ be $(t_{h_x}, q_{h_x}, m+n, \epsilon_{h_x})$-mac, $(t_{h_y}, q_{h_y}, m+n, \epsilon_{h_y})$-mac, $(t_{h_c}, q_{h_c}, m+n, \epsilon_{h_c})$-mac, respectively, and let the keyed integration function $g_2 : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be $(t_{g_2}, q_{g_2}, n+k, \epsilon_{g_2})$-mac. Then, the integrated-key constructions $x\text{-iMD}^{h_x,g_2} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^n$, $y\text{-iMD}^{h_y,g_2} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^n$, $c\text{-iMD}^{h_c,g_2} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^n$ are $(t_x, q_x, L, \epsilon_x)$-mac, $(t_y, q_y, L, \epsilon_y)$-mac, $(t_c, q_c, L, \epsilon_c)$-mac, respectively, such that:*

- $\epsilon_x = \left(\frac{q_x^2+q_x+2}{2}\right) \cdot \epsilon_{h_x} \cdot \epsilon_{g_2}, \quad \epsilon_y = \left(\frac{q_y^2+q_y+2}{2}\right) \cdot \epsilon_{h_y} \cdot \epsilon_{g_2}, \quad \epsilon_c = \left(\frac{q_c^2+q_c+2}{2}\right) \cdot \epsilon_{h_c} \cdot \epsilon_{g_2}$
- $q_i \leq \frac{q_{h_i} - e(L)}{e(L)+1}, \quad t_i = t_{h_i} - ((q_i+1) \cdot e(L) \cdot \tau_{h_i} - \tau_{h_i}) - (q_i - 1) \cdot \tau_{g_2}$

where $i \in \{x, y, c\}, \tau_{h_x}, \tau_{h_y}, \tau_{h_c}, \tau_{g_2}$ are the costs of calling h_x, h_y, h_c, g_2, respectively, and $e(L)$ is the number of m-bit blocks in a message of length L.

Proof. At this level of abstract, we can argue that x-iMD, y-iMD, c-iMD are structurally identical due to our indistinguishability result in section 6.2.1. However, in this proof we will make an explicit distinction between them and derive their advantages based on their own compression-integration composite functions. Let the composite functions h_x, h_y, h_c be $(t_{h_x}, q_{h_x}, n+m, \epsilon_{h_x})$-mac, $(t_{h_y}, q_{h_y}, n+m, \epsilon_{h_y})$-mac, $(t_{h_c}, q_{h_c}, n+m, \epsilon_{h_c})$-mac, respectively. We now build adversaries A_x, A_y, A_c against x-iMD, y-iMD, c-iMD that call adversaries B_x, B_y, B_c who attack h_x, h_y, h_c, and then derive the forging advantages of x-iMD, y-iMD, c-iMD from the advantages of these adversaries.

Adversaries. Let A_x be a $(t_{A_x}, q_{A_x}, L, \epsilon_{A_x})$-mac adversary (forger) against x-iMD that makes q_{A_x} queries of size at most L (each) to x-iMD and finds a forgery with probability ϵ_{A_x} during time at most t_{A_x}; let A_y and A_c be defined similarly for y-iMD and c-iMD. Also, let B_x be a $(t_{B_x}, q_{B_x}, n+m, \epsilon_{B_x})$-mac adversary against h_x finding forgeries with probability ϵ_{B_x} after sending q_B queries of size $m+n$ (each) to h_x, with running time at most t_{B_x}; let B_y and B_c be defined similarly for y-iMD and c-iMD. Finally, let G_{g_2} be a $(t_{G_{g_2}}, q_{G_{g_2}}, k+n, \epsilon_{G_{g_2}})$-mac adversary against g_2 that finds forgeries in g_2 with probability $\epsilon_{G_{g_2}}$ after sending $q_{G_{g_2}}$ queries of size n (each) to g_2 with running time at most $t_{G_{g_2}}$.

Unforgeability Advantage. Following the general approach due to Maurer and Sjödin [109] (also used in [25]), our unforgeability proof proceeds in 3 steps:

1. First, we propose a set of forging strategies $\mathcal{S}$ which instruct an adversary (forger) in how to forge against the x-iMD, y-iMD, c-iMD constructions.

2. Then, we prove that this set of forging strategies $\mathcal{S}$ is complete; that is, if an adversary succeeded in finding a forgery, this implies that at least one of the strategies in $\mathcal{S}$ was successful (i.e., no forgery can be created unless one of the forging strategies in $\mathcal{S}$ has been used and is successful).

3. Finally, we calculate the success probability (advantage) of the adversary in forging against x-iMD, y-iMD, c-iMD. The adversary's advantage is based on the size of $\mathcal{S}$. Obviously, the more strategies the set contains, the higher the probability that the adversary will succeed in finding a forgery that corresponds to one of those strategies. As stated earlier, if an adversary found a forgery, it must have been generated by one of those strategies, so a forging strategies set $\mathcal{S}$ with minimal size means that the adversary has less chance to succeed.

To simplify the following discussion, we abstract the x-iMD, y-iMD, c-iMD constructions and their adversaries. Essentially, when we give x-iMD, y-iMD, c-iMD access to the composite functions h_x, h_y, h_c, these constructions become structurally identical. Thus, we will denote the x-iMD, y-iMD, c-iMD constructions by iMD and the adversaries attacking them A_x, A_y, A_c by A. Similarly, we will refer to the composite functions h_x, h_y, h_c by h and the adversaries attacking them B_x, B_y, B_c by B. We will later decompose and state the bounds for each construction.

In order to obtain the forging bound of iMD, we derive the forging advantage of A (who attacks iMD) in terms of B (who attacks h). Specifically, B picks a strategy $s_i \in \mathcal{S}$, runs A according to s_i and obtains A's forging advantage in terms of its own (i.e., B's) forging advantage. Let the number of calls to h required to process all the q_A queries that A sends during its running time t_A be ℓ_A (i.e., $\ell_A \leq q_A \cdot e(L)$, where $e(L)$ is the number of blocks in a message of length L), then the set of forging strategies is defined as follows:

$$\mathcal{S} = \left\{ \{(i, w) : w \in Rng(h)\} \cup \{(i, j) : 1 \leq j < i \leq \ell_A\} \cup \{(\ell_A, z) : z \in Rng(h)\} \right\}$$

where $i, j \in \{1, \ldots, \ell_A\}$ and $Rng(h)$ is the range of h which is $\{0,1\}^n$. In particular, $\mathcal{S}$ contains 3 types of strategies:

1. $(i, w), i \in \{1, \ldots, \ell_A\}, w \in \{0,1\}^n$: in these strategies B predicts that the i-th application of h will return w.

2. $(i, j), 1 < j < i, i, j \in \{1, \ldots, \ell_A\}$: here B predicts that there is a collision between two internal applications of h, that is, the output of some earlier application (the j-th) of h equals the output of the i-th application of h for different inputs.

3. $(\ell_A, z), z \in \{0,1\}^n$: this is a single strategy where B predicts the output of A (the ℓ_A-th application of h), the prediction in this case is z. This strategy is similar to the naïve strategy in [109].

Once the simulation is started, B at some point forcefully terminates the simulation according to the chosen forging strategy $s_i \in \mathcal{S}$, then returns a forgery message M and a tag T, such that $\text{iMD}(M) = T$, which are both decided based on s_i. Let $M^{(i)}$ denote the $m + n$ bit string that would be the input to the i-th application of h, precisely $M^{(i)} = m_i || y_{i-1}$, where m_i is the i-th block of a query and y_{i-1} is the output of the $(i-1)$-th application of h. In the strategy (i, w), B stops the simulation right before A's i-th query to h, and returns $(M^{(i)}, w)$. Similarly, in the strategy (i, j), B stops the simulation right before A's i-th query and returns $(M^{(i)}, y_j)$, where y_j is the output of a previous (in this case the j-th) application of h (i.e., $j < i$), that is $y_j = h(M^{(j)})$. Finally, if strategy (ℓ_A, z) was chosen, B waits until the very last application of h and right before processing $M^{(\ell_A)}$, it terminates the simulation and returns (M^{ℓ_A}, z), where z is a prediction of $\text{iMD}(m_1 || m_2 || \ldots || m_{\ell_A})$.

Lemma 6.3.2 (Completeness of $\mathcal{S}$). *The set of forging strategies $\mathcal{S}$ as defined above is complete in the sense that if a forging adversary A is successful in finding a forgery against iMD, then at least one strategy in $\mathcal{S}$ is successful.*

Proof. Suppose that there is an adversary A sending q_A queries to iMD, each query of length at most L, then with probability ϵ_A, it finds a forgery. Let the number of h applications required to process the q_A queries be ℓ_A, that is $q_1 || q_2 || \ldots || q_A = m_1 || m_2 || \ldots || m_{\ell_A}$ where $m_1 || m_2 || \ldots || m_{\ell_A}$ is the concatenation of the blocks of all queries (note that a single query may consist of more than 1 block). Then A either finds a forgery in the last application of h, or some internal application of h. If A finds a forgery in the last application of h, then strategy (ℓ_A, z) is successful. Otherwise, A finds a forgery in an internal application of h, in which case there are two possibilities:

1. A predicts a collision between two internal applications of h, specifically, the i-th and j-th ones, where $1 \leq j < i$ and $i, j \in \{1, 2, \ldots, \ell - 1\}$, then A returns the output of the j-th application of h as a forgery tag and what would be the input of the i-th application of h as the forgery message, that is $(M^{(i)}, y_j)$, where y_j is the output of the j-th application of h, and $M^{(i)}$ is the input to the i-th application of h. In this case, strategy (i, j) succeeds.

2. A predicts the output of the i-th application of h to be, say, $w \in \{0,1\}^n$, where $i \in \{1, 2, \ldots, \ell - 1\}$, and returns its prediction value w as a forgery tag, while returning what would be the input to the i-th application of h as the forgery message, that is (M^i, w). In this case, strategy (i, w) succeeds.

For a message consisting of only one block, strategy (ℓ_A, z) succeeds where, in this case, $\ell_A = 1$. Therefore, it is clear that if A finds a forgery in iMD, then at least one of the strategies in $\mathcal{S}$ is successful. □

In the discussion of the forging strategies and the proof of Lemma 6.3.2, we skipped a technicality that we elaborate on here, that is B's success probability of generating a valid tag when it outputs a forgery according to the various forging strategies in $\mathcal{S}$. We know that B finds a forgery in h with success probability ϵ_B, but recall that iMD further finalises its output with a call to g_2, so in order for a forgery (M, T) to be valid (i.e., $\text{iMD}(M) = T$), the tag T must have been constructed in such a way that this finalisation call of g_2 (beside the previous h calls) is accounted for. That is, if B forcefully terminates the simulation of A before A actually finishes processing the whole message (and B will always do), B should also call adversary G_{g_2} to finalise T, which returns a valid tag with probability $G_{\epsilon_{g_2}} = \epsilon_{g_2}$. Also, in order to obtain the advantage of B in terms of the advantage of A, the latter has to be divided by the size of the forging strategies $|\mathcal{S}|$ that B adopts in its forging game. Thus,

$$\mathbf{Adv}^{\text{mac}}_{B} \cdot \mathbf{Adv}^{\text{mac}}_{G_{g_2}} = \frac{\mathbf{Adv}^{\text{mac}}_{A}}{|\mathcal{S}|}$$

then solving for A, we get

$$\mathbf{Adv}^{\text{mac}}_{A} = |\mathcal{S}| \cdot \mathbf{Adv}^{\text{mac}}_{B} \cdot \mathbf{Adv}^{\text{mac}}_{G_{g_2}}$$

The size of $\mathcal{S}$ is the number of strategies it contains, and is calculated as follows: strategy (i, w) contributes ℓ_A pairs (since $i \in \{1, 2, \ldots, \ell_A\}$), strategy (i, j) contributes $(\ell_A^2 - \ell_A)/2$ pairs [109], and strategy (ℓ_A, z) contributes a single pair, thus:

$$|\mathcal{S}| = \ell_A + \frac{\ell_A^2 - \ell_A}{2} + 1 = \frac{2\ell_A + \ell_A^2 - \ell_A}{2} + 1 = \frac{\ell_A^2 + \ell_A + 2}{2}$$

Now we can obtain the success bounds of A_x, A_y, A_c based on the success probability of B_x, B_y, B_c and G_{g_2} as well as the size of $\mathcal{S}$. For x-iMD:

$$\mathbf{Adv}^{\text{mac}}_{B_x} \cdot \mathbf{Adv}^{\text{mac}}_{G_{g_2}} = \frac{\mathbf{Adv}^{\text{mac}}_{A_x}}{(\ell_{A_x}^2 + \ell_{A_x} + 2)/2}$$

then solving for ϵ_{A_x}, we get:

$$\begin{aligned}\mathbf{Adv}^{\text{mac}}_{A_x} &= \left(\frac{\ell^2_{A_x}+\ell_{A_x}+2}{2}\right)\cdot\mathbf{Adv}^{\text{mac}}_{B_x}\cdot\mathbf{Adv}^{\text{mac}}_{G_{g2}}\\ &= \left(\frac{\ell^2_{A_x}+\ell_{A_x}+2}{2}\right)\cdot\mathbf{Adv}^{\text{mac}}_{h_x}\cdot\mathbf{Adv}^{\text{mac}}_{g_2} = \left(\frac{\ell^2_{A_x}+\ell_{A_x}+2}{2}\right)\cdot\epsilon_{h_x}\cdot\epsilon_{g_2}\end{aligned}$$

The advantages of y-iMD and c-iMD follow.

Queries. Recall that in the iMD constructions, B is an adversary against h, but in order to find a forgery in h, B runs A, then A calls h repeatedly to process its q_A queries and returns a forgery with probability ϵ_A. Here we will assume that the adversary needs *at least* q_A queries to find a forgery, meaning that we would expect a forgery in the query q_A+1 (note that some authors assume that the q_A-th query is the forgery query). In the worst case, A will need to call h to process all message blocks in every query $q_i \in \{1,2,\ldots,q_A\}$. Let the number of blocks in a single query of length at most L be denoted by $e(L)$, then the number of queries made to h should be $\leq q_A \cdot e(L)$. Clearly, in the worst case, A will send all its queries with the maximum length L and then h will be queried by A at most $e(L)$ for every query sent, thus the total times A queries h is $q_A \cdot e(L)$, but we also need to account for the forgery query (when it is found). In the worst case, the forging strategy (ℓ_A, z) will succeed and a forgery will be found in the last application of h, this gives a further $e(L)-1$ queries to h (we deduct 1 from $e(L)$ because the last application of h is not actually made when a forgery is found, it is rather predicted according to the forging strategy, which is the (ℓ_A, z) strategy), this gives $q_B \leq q_A \cdot e(L) + (e(L)-1)$. Finally, recall that there is a finalisation call to g_2 made at the end of every query (including the forgery query), this adds further q_A+1 queries made to g_2. Thus,

$$\begin{aligned}q_B &\leq q_A \cdot e(L) + (e(L)-1) + q_A + 1\\ &\leq q_A(e(L)+1) + (e(L)-1) + 1\end{aligned}$$

and solving for q_A, we get:

$$q_A \leq \frac{q_B-(e(L)-1)-1}{e(L)+1} \leq \frac{q_B-e(L)}{e(L)+1}$$

Running time. Since B runs A once and terminates when A terminates (or when it forcefully terminates the simulation), it was suggested in [109] that $t_B \approx t_A$, but in our

case we have two different functions that are being accessed while processing a query, namely the composite function h and the finalisation function g_2. Since the efficiency of these two functions are likely to differ, here we derive the running time of B and A in terms of both h and g_2. Let τ_h denote the cost of calling h and τ_{g_2} denote the cost of calling g_2, then:

$$t_B = t_A + ((q_A + 1) \cdot e(L) \cdot \tau_h - \tau_h) + (q_A - 1) \cdot \tau_{g_2}$$

and solving for t_A, we get:

$$t_A = t_B - ((q_A + 1) \cdot e(L) \cdot \tau_h - \tau_h) - (q_A - 1) \cdot \tau_{g_2}$$

Basically, we count how many times h and g_2 are called during the time A was running, then multiply these by τ_h and τ_{g_2}, respectively. In the worst case, A will process all its queries and will only find a forgery in the last block of the last query (query $q_A + 1$), in which case strategy (ℓ_A, z) succeeds. We observe that h is being called $q_A \cdot e(L)$ times, but we also need to add further $e(L) - 1$ calls to h to account for the query that will return a forgery. We deduct 1 from $e(L) - 1$ because the message block at which the forgery is found is not further processed by h, but instead its output, which will form the forgery tag, is predicted according to the adopted forging strategy, which is, in the worst case, the (ℓ_A, z) strategy. Unlike h, g_2 is called only once per query, so we have $q_A \cdot \tau_{g_2}$, again g_2 is not called at the forgery query, but instead A predicts the forgery tag to a value that has already been processed by g_2, thus we multiply $(q_A - 1) \cdot \tau_{g_2}$. The running times for A_x, A_y, A_c are identical. □

6.4 Summary

In this chapter we showed that the iMD constructions (namely, x-iMD, y-iMD, c-iMD), proposed previously in chapter 4, are indistinguishable from their variants in the dedicated-key setting (where the latter use keyed compression functions), then based on the indifferentiability result in chapter 5, we showed that the iMD constructions are also trivially indistinguishable from Pseudorandom Functions (PRF). We used this indistinguishability result to further prove that the iMD constructions are unforgeable when treated as MACs (in the secretly key setting, where keys are kept private). Unforgeability means that there is no adversary able to forge a valid message-tag pair such that when the tag is processed by a MAC algorithm based on one of the iMD constructions, it produces the corresponding (provided) tag. This chapter concludes the contributions of the thesis.

Chapter 7

Conclusion and Future Work

Interest in cryptographic hash functions has recently spiked making it one of the most highly active research areas in cryptography. Consequently, the literature has expanded significantly over the last few years to the point where some considered it to be unmanageable. In this thesis, we first tried to provide an up-to-date survey of the current state of the art of cryptographic hash functions and then proceeded to propose a new hash function class, which we called the *integrated-key* setting. We primarily proposed the integrated-key setting as a low-effort mechanism to convert (and thus strengthen) keyless hash functions to keyed ones. The main design goal of this approach is to make this conversion process as transparent as possible by lifting the burden of having to modify the internal structure of the underlying keyless primitives of the keyless hash function to accommodate the key input. However, that does not mean that the integrated-key setting may not be considered a dedicated design approach in its own right, which hash functions that are being built from scratch can still adopt. We first proposed a few Merkle-Damgård based integrated-key hash functions. Security analyses of these integrated-key functions have then extended over most of the thesis which constitute our main contributions. In chapter 4, we proved that these constructions are collision resistant, pre-image resistant and 2nd pre-image resistant. Then, in chapter 5 we showed that they are also indifferentiable from $\mathcal{RO}$. Finally, in chapter 6 we showed that these constructions are indistinguishable from their variants in the dedicated-key setting and then used this result to prove that they are further indistinguishable from Pseudorandom Functions (PRF) and unforgeable when they are instantiated as Message Authentication Codes (MAC) in the secret key setting. Throughout the thesis, we hope that we succeeded in demonstrating that the integrated-key setting is not only theoretically interesting but also practically relevant. In this chapter, the thesis concludes with final remarks and brief discussions about how our work can be extended.

7.1 Preservation of Other Properties

While in this thesis we provided proofs for the most common properties that the majority of hash functions applications require, as we showed in chapter 2, there are many other hash functions properties in the literature. It would certainly be interesting to see whether our proposed x-iMD, y-iMD, c-iMD constructions preserve these other properties. One such property is Target Collision Resistance (TCR), which some applications may require. The term Target Collision Resistance was first coined by Bellare and Rogaway [27] for the classical notion of universal one way hash function (UOWHF) of Naor and Young [116]. TCR is a weaker notion than collision resistance, and thus any CR hash function, is also a TCR [135]. That is, if we can prove that a hash function is CR given its compression function is CR, then that hash function is also TCR. However, in some settings [27], it is desirable to have a TCR hash function based on the assumption that its compression function is only TCR (not necessarily CR). We expect that the plain iMD constructions are *not* TCR since there are strong evidence [113] that for Merkle-Damgård like constructions (which our iMD constructions are variants of), the Sh construction [141] (which uses more key material than our iMD constructions) is optimal. Obviously, an easy fix is to adopt the Sh keying schedule in the iMD constructions, which is easy and does not require any modification to the constructions since in Sh the key mask is merely XORed with the chaining variables.

7.2 Unified Message Preservation

We will also be interested in studying other properties that could have more practical relevance. An example of such class of properties is what we call "Unified-Message Preservation" (UMP). This class builds on the traditional collision, pre-image, 2nd pre-image resistance notions and their "always" and "everywhere" variants proposed by Rogaway and Shrimpton [135]. Basically, the UMP properties impose more restrictions on the adversary A and further requires that the returned message has a particular length because this is what would have the most practical relevance. This means that the UMP properties are weaker than the conventional properties due to Rogaway and Shrimpton, but they have more practical applications, and indeed making proofs based on weaker assumptions is one of the goals in cryptography (that is, the weaker the assumptions you make on your primitive, the more practical it is). For example, finding a pre-image or 2nd pre-image of length different than the original message is theoretically interesting but has little practical effect in most cases since the 2nd message that the adversary finds may not replace the original one in some applications.

Similarly, in collision resistance, if an adversary outputs two messages with different lengths, he cannot use them interchangeably in all applications, potentially defeating the practicality of the attack. Formally, the advantage definitions of the UMP variants of collision, pre-image, 2nd pre-image and their "always" and "everywhere" variants are as follows:

$$\mathbf{Adv}^{\text{u-cr}}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} K \xleftarrow{\$} \mathcal{K}; (M, M') \xleftarrow{\$} A(K) : \\ M \neq M' \wedge |M| = |M'| \wedge H_K(M) = H_K(M') \end{array}\right]$$

$$\mathbf{Adv}^{\text{u-pre}[m]}_{H_K}(A) = \Pr\left[\begin{array}{l} K \xleftarrow{\$} \mathcal{K}; M \xleftarrow{\$} \{0,1\}^m; Y \leftarrow H_K(M); M' \xleftarrow{\$} A(K, Y) : \\ |M'| = |M| \wedge H_K(M') = Y \end{array}\right]$$

$$\mathbf{Adv}^{\text{u-aPre}[m]}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} (K, St) \xleftarrow{\$} A; M \xleftarrow{\$} \{0,1\}^m; Y \leftarrow H_K(M); \\ M' \xleftarrow{\$} A(Y, St) : |M| = |M'| \wedge H_K(M') = Y \end{array}\right]$$

$$\mathbf{Adv}^{\text{u-sec}[m]}_{H_K}(A) = \Pr\left[\begin{array}{l} K \xleftarrow{\$} \mathcal{K}; M \xleftarrow{\$} \{0,1\}^m; M' \xleftarrow{\$} A(K, M) : \\ M \neq M' \wedge |M| = |M'| \wedge H_K(M) = H_K(M') \end{array}\right]$$

$$\mathbf{Adv}^{\text{u-aSec}[m]}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} (K, St) \xleftarrow{\$} A; M \xleftarrow{\$} \{0,1\}^m; M' \xleftarrow{\$} A(M, St) : \\ M \neq M' \wedge |M| = |M'| \wedge H_K(M) = H_K(M') \end{array}\right]$$

$$\mathbf{Adv}^{\text{u-eSec}}_{H_K}(\mathrm{A}) = \Pr\left[\begin{array}{l} (M, St) \leftarrow A; K \xleftarrow{\$} \mathcal{K}; M' \xleftarrow{\$} A(K, St) : \\ M \neq M' \wedge |M| = |M'| \wedge H_K(M) = H_K(M') \end{array}\right]$$

where M is a message, $K \in \mathcal{K}$ is a key, H_K is a keyed hash function and Y is the hash value of the message M. Note that UMP cannot be applied on the ePre property because in ePre A generates the hash value Y uniformly at random, thus Y may not correspond to a particular message that A needs to match its length with the length of the pre-image message A will generate later. Strengthened variants of Rogaway's and Shrimpton's properties also exist [135] as discussed in section 2.7, which we can too derive UMP variants for. Clearly, it will be interesting to study the implications and separations among all these variants. There are different types of implications and separations (e.g., unconditional separation etc.), thus with this new class of properties we expect to be able to derive yet other types of implications and separations which then we can use to gain more understanding of the relations among the different properties. Another interesting problem is to investigate whether the preservation of a particular property by a particular construction prevents the preservation of another. In other words, it will be interesting if we found that a particular construction *cannot* preserve two (or more) particular properties at the same time.

7.3 Keyless Hash Functions from Keyed Ones

In this thesis, we introduced and analysed a way of transforming keyless hash functions into keyed ones without tampering with the compression function. This has been mainly motivated by the security benefits that the keyed setting provides. However, it might be also interesting to study the other way round (converting keyed hash functions into keyless ones). This will be relevant for applications that require keyless hash functions and cannot be adapted to use keyed ones. In this case, if we found a way to covert a keyed hash function into a keyless variant that still enjoys adequate security margin, we can immediately use such keyed variant in those applications. One easy solution is to fix the key input to 0 at the keyless setting, e.g., HAIFA [66], but this does not completely solve the efficiency waste introduced by the key input. A better approach is, then, to expand the message blocks using the key input. For example, conventional keyed hash functions $C^h : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$ use a keyed compression function $h : \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$ to process the message in m-bit blocks. If the k-bit key input is not needed, then the message block can be extended to $m + k$ bit which will both convert C^h to keyless and improve its efficiency (since each compression iteration now processes more message bits, which means that we would generally need fewer compression function calls). This, however, depends on the way h mixes K with the rest of its input and whether mixing part of M in this manner is acceptable.

7.4 Integration Function Design

While proving the various security properties of our iMD constructions, we made several assumptions about the integration function, such as being collision resistant, $\mathcal{RO}$ etc., then our proofs hold as long as our assumptions also hold, and this is a standard proof approach in cryptography. Although in section 4.3 we recommended the PPC[1] design paradigm [133, 63] as a potential integration function design, it worth investigating other approaches. It is important that the chosen function is sufficiently efficient for all practical applications and strictly more efficient than the compression function, otherwise at least double the work will be required to process a single message block. Although the integration function is generally conceived as a compressing primitive, it can indeed be built from non-compressing primitives [142] as long as the efficiency is not severely hindered. Generally, the integration function should preserve the same properties as those preserved by the compression function, but this, in most cases, depends on the construction and where it places the integration function.

[1] In the Prepend-Permute-Chop paradigm (PPC), the inputs to the function are concatenated, permuted and then several bits are chopped before returning the output.

7.5 Pre Proof in the Standard Model

In chapter 4 we provided the Pre (pre-image resistance) proof of the iMD constructions in the Random Oracle Model (ROM) where we modelled both the compression functions and the integration functions as $\mathcal{RO}$s. If we adopt the PPC paradigm (as discussed in section 4.3) for the integration function (which is proven indifferentiable from $\mathcal{RO}$), then the PPC-based integration function can be thought of as being part of the integrated-key setting. On the other hand, we do not have control over the compression function and assuming that it is indifferentiable from $\mathcal{RO}$ is a strong assumption. However, we adopted the ROM approach in our Pre proof because it is inherently difficult to develop such proofs in the standard model. In fact, it appears that proofs of the other Pre variants (especially aPre) are also as difficult in the standard model. Thus, an extension to this work is to investigate the plausibility of devising a generic Pre proof approach in the standard model, which hopefully other integrated-key (and even dedicated-key) hash functions can adopt while arguing about their own Pre security.

7.6 Final Remarks

One of the classical (and challenging) problems in the literature of cryptographic hash functions is how to convert keyless hash functions into keyed ones since the latter has more advantages in terms of security over the former. In this thesis, we developed and defined a new hash function design approach in which keyless hash functions are transparently transformed into families of hash functions by keying them, we call this approach the integrated-key setting. Although this approach introduced an efficiency loss, this is unfortunately inevitable in any keyed setting since in this setting an extra input (i.e., the key) is introduced and needs to be appropriately accommodated somehow. However, in the integrated-key setting, the amount of efficiency loss highly depends on the implementation of the integration function; this potentially makes the integration function a customisable primitive and thus gives more freedom for implementers to adjust their implementations based on the requirement of the application to which the hash function is targeted. We proposed several integrated-key hash functions and proved that they preserve the most commonly required properties, namely collision resistance, pre-image resistance, 2nd pre-image resistance, indifferentiable from $\mathcal{RO}$, indistinguishable from Pseudorandom Functions (PRF) and unforgeable when used as MACs. We also showed that the integrated-key hash functions are indistinguishable from their variants in the dedicated-key setting, which means that proofs in the dedicated-key setting is reducible to the integrated-key one.

Bibliography

[1] ISO/IEC 9797-2 – Information technology – Security techniques – Message Authentication Codes (MACs) – Part 2: Mechanisms using a dedicated hash-function. 31, 39

[2] S. Al-Kuwari. Engineering Aspects of Hash Functions. In *International Conference on Security and Management (SAM '11)*, 2011. 10, 139

[3] S. Al-Kuwari. Integrated-key Hash Functions: How to Constructing Keyed Hash Functions from Keyless Ones, 2011. (manuscript under review). 10

[4] S. Al-Kuwari. On the Indifferentiability of Integrated-key Hash Functions, 2011. (manuscript under review). 10

[5] S. Al-Kuwari, J. Davenport, and R. Bradford. Cryptographic Hash Functions: Recent Design Trends and Security Notions. In *Short Paper Proceedings of Inscrypt '10*, pages 133–150. Science Press of China, 2010. (Full version available at `eprint.iacr.org/2011/565`). 10, 11, 29

[6] S. Al-Kuwari, J. Davenport, and R. Bradford. Cryptographic Hash Functions: Recent Design Trends and Security Notions. Cryptology ePrint Archive, Report 2011/565, 2011. (`eprint.iacr.org/2011/565`). 10, 11, 29

[7] J. An and M. Bellare. Constructing VIL-MACs from FIL-MACs: Message Authentication under Weakened Assumptions. In *Crypto '99*, volume 1666 of *LNCS*, pages 252–269. Springer-Verlag, 1999. 107

[8] R. Anderson, E. Biham, and L. Knudsen. *Serpent: A Proposal for the Advanced Encryption Standard*, 1997. (`www.cl.cam.ac.uk/~rja14/serpent`). 140

[9] E. Andreeva, C. Bouillaguet, P.-A. Fouque, J. Hoch, J. Kelsey, A. Shamir, and S. Zimmer. Second Preimage Attacks on Dithered Hash Functions. In *Eurocrypt '08*, volume 4965 of *LNCS*, pages 270–288. Springer-Verlag, 2008. 41

[10] E. Andreeva, O. Dunkelman, C. Bouillaguet, and J. Kelsey. Herding, Second Preimage and Trojan Message Attacks Beyond Merkle-Damgård. In *SAC '09*, volume 5867 of *LNCS*, pages 393–414. Springer-Verlag, 2009. 37

[11] E. Andreeva, B. Mennink, and B. Preneel. Security Properties of Domain Extenders for Cryptographic Hash Functions. *Journal of Information Processing Systems*, 6(4):453–480, 2010. 39

[12] E. Andreeva, B. Mennink, and B. Preneel. Security Reductions of the Second Round SHA-3 Candidates. In *ISC '11*, volume 6531 of *LNCS*, pages 39–53. Springer-Verlag, 2011. 30

[13] E. Andreeva, B. Mennink, and B. Preneel. The Parazoa Family: Generalizing the Sponge Hash Functions. Cryptology ePrint Archive, Report 2011/028, 2011. (eprint.iacr.org/2011/028). 43

[14] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton. Seven-Property-Preserving Iterated Hashing: ROX. In *Asiacrypt '07*, volume 4833 of *LNCS*, pages 130–146. Springer-Verlag, 2007. 27, 55, 61

[15] E. Andreeva and B. Preneel. A Three-Property-Secure Hash Function. In *SAC '09*, volume 5381 of *LNCS*, pages 228–244. Springer-Verlag, 2009. 27, 51, 55

[16] D. Augot, M. Finiasz, P. Gaborit, S. Manuel, and N. Sendrier. *SHA-3 Proposal: FSB*, 2008. (www-rocq.inria.fr/secret/CBCrypto). 47

[17] D. Augot, M. Finiasz, and N. Sendrier. A Fast Provably Secure Cryptographic Hash Function. Cryptology ePrint Archive, Report 2003/230, 2003. (eprint.iacr.org/2003/230). 47

[18] D. Augot, M. Finiasz, and N. Sendrier. A Family of Fast Syndrome Based Cryptographic Hash Functions. In *Mycrypt '05*, volume 3715 of *LNCS*, pages 64–83. Springer-Verlag, 2005. 47

[19] M. Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In *Crypto '06*, volume 4117 of *LNCS*, pages 602–619. Springer-Verlag, 2006. 107

[20] M. Bellare, A. Boldyreva, and A. Palacio. An Uninstantiable Random-Oracle-Model Scheme for a Hybrid-Encryption Problem. In *Eurocrypt '04*, volume 3027 of *LNCS*, pages 171–188. Springer-Verlag, 2004. 71

[21] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In *Crypto '96*, volume 1109 of *LNCS*, pages 1–15. Springer-Verlag, 1996. 39, 106

[22] M. Bellare, R. Gurin, and P. Rogaway. XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions. In *Crypto '95*, volume 963 of *LNCS*, pages 15–28. Springer-Verlag, 1995. 107

[23] M. Bellare and D. Micciancio. A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost. In *Eurocrypt '97*, volume 1233 of *LNCS*, pages 163–192. Springer-Verlag, 1997. 44

[24] M. Bellare and T. Ristenpart. Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In *Asiacrypt '06*, volume 4284 of *LNCS*, pages 299–314. Springer-Verlag, 2006. 21, 25, 38, 41, 43, 51

[25] M. Bellare and T. Ristenpart. Hash Functions in the Dedicated-Key Setting: Design Choices and MPP Transforms. In *ICALP '07*, volume 4596 of *LNCS*, pages 399–410. Springer-Verlag, 2007. 27, 31, 40, 42, 43, 50, 53, 112, 114

[26] M. Bellare and P. Rogaway. Random Oracles are Practical: a Paradigm for Designing Efficient Protocols. In *CCS '93*, pages 62–73, 1993. 17, 71

[27] M. Bellare and P. Rogaway. Collision-resistant Hashing: Towards Making UOWHF's Practical. In *Cryptgo '97*, volume 1294 of *LNCS*, pages 470–484. Springer-Verlag, 1997. 26, 44, 120

[28] M. Bellare and P. Rogaway. Code-Based Game-Playing Proofs and the Security of Triple Encryption. Cryptology ePrint Archive, Report 2004/331, 2004. (`eprint.iacr.org/2004/331`). 21, 76

[29] M. Bellare and Tadayoshi. Hash Function Balance and its Impact on Birthday Attacks. In *Eurocrypt '04*, volume 3027 of *LNCS*, pages 401–418. Springer-Verlag, 2004. 7, 11

[30] R. Benadjila, O. Billet, S. Gueron, and M. Robshaw. The Intel AES Instructions Set and the SHA-3 Candidates. In *Asiacrypt '09*, volume 5912 of *LNCS*, pages 162–178. Springer-Verlag, 2009. 142

[31] D. Bernstein. *CubeHash Specification*, 2008. (`cubehash.cr.yp.to`). 46

[32] D. J. Bernstein, T. Lange, C. Peters, and P. Schwabe. Really Fast Syndrome-based Hashing. Cryptology ePrint Archive, Report 2011/074, 2011. (eprint.iacr.org/2011/074). 47

[33] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge Functions. In *ECRYPT Hash Workshop*, 2007. 16, 43

[34] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the Indifferentiability of the Sponge Construction. In *Eurocrypt '08*, volume 4965 of *LNCS*, pages 181–197. Springer-Verlag, 2008. 44

[35] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. *The Keccak Sponge Function Family*, 2008. (keccak.noekeon.org). 43

[36] E. Biham and A. Shamir. Differential Cryptanalysis of the Data Encryption Standard. In *Crypto '90*, volume 537 of *LNCS*, pages 2–21. Springer-Verlag, 1990. 51

[37] J. Black, M. Cochran, and T. Shrimpton. On the Impossibility of Highly-Efficient Blockcipher-Based Hash Functions. *Journal of Cryptology*, 22:311–329, 2009. 46

[38] J. Black, P. Rogaway, and T. Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In *Crypto '02*, volume 2442 of *LNCS*, pages 320–335. Springer-Verlag, 2002. 45, 46

[39] J. Buchmann and S. Paulus. A One Way Function Based on Ideal Arithmetic in Number Fields. In *Crypto '97*, volume 1294 of *LNCS*, pages 385–394. Springer-Verlag, 1997. 47

[40] R. Canetti, O. Goldreich, and S. Halevi. The Random Oracle Methodology, Revisited. *Journal of the ACM*, 51(4):557–594, 1998. 17, 71

[41] R. Canetti, O. Goldreich, and S. Halevi. On the Random-Oracle Methodology as Applied to Length-Restricted Signature Schemes. In *TCC ' 04*, volume 2951 of *LNCS*, pages 40–57. Springer-Verlag, 2004. 71

[42] D. Chang. Preimage Attacks on CellHash, SubHash and Strengthened Versions of CellHash and SubHash. Cryptology ePrint Archive, Report 2006/412, 2006. (eprint.iacr.org/2006/412). 48

[43] D. Chang, S. Lee, M. Nandi, and M. Yung. Indifferentiable Security Analysis of Popular Hash Functions with Prefix-Free Padding. In *ASIACRYPT '06*, volume 4284 of *LNCS*, pages 283–298. Springer-Verlag, 2006. 72

[44] D. Charles, K. Lauter, and E. Goren. Cryptographic Hash Functions from Expander Graphs. *Journal of Cryptology*, 22(1):93–113, 2007. 47

[45] D. Chaum, E. van Heijst, and B. Pfitzmann. Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer. In *Crypto '91*, volume 576 of *LNCS*, pages 470–484. Springer-Verlag, 1991. 47

[46] S. Contini, A. Lenstra, and R. Steinfeld. VSH, an Efficient and Provable Collision-Resistant Hash Function. In *Eurocrypt '06*, volume 4004 of *LNCS*, pages 165–182. Springer-Verlag, 2006. 47

[47] S. Contini, R. Steinfeld, J. Pieprzyk, and K. Matusiewicz. A Critical Look at Cryptographic Hash Function Literature. In *ECRYPT Hash Function Workshop*, 2007. 15, 31

[48] J. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In *Crypto '05*, volume 3621 of *LNCS*, pages 430–448. Springer-Verlag, 2005. 17, 19, 21, 22, 25, 34, 38, 39, 41, 54, 70, 71, 72, 76, 106

[49] J.-S. Coron and A. Joux. Cryptanalysis of a Provably Secure Cryptographic Hash Function. Cryptology ePrint Archive, Report 2004/013, 2004. (`eprint.iacr.org/2004/013`). 47

[50] J.-S. Coron, J. Patarin, and Y. Seurin. The Random Oracle Model and the Ideal Cipher Model are Equivalent. In *Crypto '08*, volume 5157 of *LNCS*, pages 1–20. Springer-Verlag, 2008. 72

[51] J. Daemen, R. Govaerts, and J. Vandewalle. A Framework for the Design of One-Way Hash Functions Including Cryptanalysis of Damgård's One-Way Function Based on a Cellular Automaton. In *Asiacrypt '91*, volume 739 of *LNCS*, pages 82–96. Springer-Verlag, 1991. 48

[52] J. Daemen, R. Govaerts, and J. Vandewalle. A Hardware Design Model for Cryptographic Algorithms. In *ESORICS '92*, volume 648 of *LNCS*, pages 419–434. Springer-Verlag, 1992. 48

[53] J. Daemen and V. Rijmen. *AES Proposal: Rijndael*, 1997. 140

[54] I. Damgåard. Collision Free Hash Functions and Public Key Signature Schemes. In *Eurocrypt '87*, volume 304 of *LNCS*, pages 203–216. Springer-Verlag, 1987. 13, 14

[55] I. Damgård. A Design Principle for Hash Functions. In *Crypto '89*, volume 435 of *LNCS*, pages 416–427. Springer-Verlag, 1989. 15, 19, 31, 32, 33, 44, 47, 48, 54, 56, 70

[56] I. Damgård, L. Knudsen, and S. Thomsen. DAKOTA – Hashing from a Combination of Modular Arithmetic and Symmetric Cryptograph. In *ACNS '08*, volume 5037 of *LNCS*, pages 144–155. Springer-Verlag, 2008. 47

[57] Q. Dang. NIST Special Publication 800-106: Randomized Hashing for Digital Signatures. Technical report, National Institute of Standards and Technology, 2009. 40

[58] R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999. 35

[59] S. Deng, Y. Li, and D. Xiao. Analysis and Improvement of a Chaos-based Hash Function Construction. *Communications in Nonlinear Science and Numerical Simulation*, 15(5):1338–1347, 2009. 48

[60] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. 1, 29

[61] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin. Randomness Extraction and Key Derivation Using the CBC, Cascade and HMAC Modes. In *Crypto '04*, volume 3152 of *LNCS*, pages 494–510. Springer-Verlag, 2004. 25

[62] Y. Dodis and P. Puniya. Getting the Best Out of Existing Hash Functions; or What if We Are Stuck with SHA? In *ACNS '08*, volume 5037 of *LNCS*, pages 156–173. Springer-Verlag, 2008. 25

[63] Y. Dodis, L. Reyzin, R. Rivest, and E. Shen. Indifferentiability of Permutation-Based Compression Functions and Tree-Based Modes of Operation, with Applications to MD6. In *FSE '09*, volume 5665 of *LNCS*, pages 104–121. Springer-Verlag, 2009. 54, 122

[64] Y. Dodis, T. Ristenpart, and T. Shrimpton. Salvaging Merkle-Damgård for Practical Applications. In *Eurocrypt '09*, volume 5479 of *LNCS*, pages 371–388. Springer-Verlag, 2009. 22

[65] Y. Dodis and J. Steinberger. Message Authentication Codes from Unpredictable Block Ciphers. In *Crypto '09*, volume 5157 of *LNCS*, pages 267–285. Springer-Verlag, 2009. 23

[66] O. Dunkelman and E. Biham. A Framework for Iterative Hash Functions – HAIFA. In *2nd NIST Cryptographich Hash Workshop*, 2006. 40, 122

[67] O. Dunkelman and B. Preneel. Generalizing the Herding Attack to Concatenated Hashing Schemes. In *ECRYPT Hash Function Workshop*, 2007. 37

[68] L. Duo and C. Li. Improved Collision and Preimage Resistance Bounds on PGV Schemes. Cryptology ePrint Archive, Report 2006/462, 2006. (eprint.iacr.org/2006/462). 46

[69] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. *The Skein Hash Function*, 2008. (www.skein-hash.info). 45, 51, 140, 141

[70] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & Sons, 2003. 34

[71] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography Engineering: Design, Principles and Practical Applications*. Wiley Publishing, 2010. 24

[72] P.-A. Fouque and G. Leurent. Cryptanalysis of a Hash Function Based on Quasi-cyclic Codes. In *CT-RSA '08*, volume 4964 of *LNCS*, pages 19–35. Springer-Verlag, 2008. 47

[73] P. Gauravaram and J. Kelsey. Linear-XOR and Additive Checksums Dont Protect Damgrd-Merkle Hashes from Generic Attacks. In *CT-RSA '08*, volume 4964 of *LNCS*, pages 36–51. Springer-Verlag, 2008. 38

[74] P. Gauravaram, J. Kelsey, L. R. Knudsen, and S. S. Thomsen. On Hash Functions Using Checksums. *International Journal of Information Security*, 9(2):137–151, 2010. 38

[75] P. Gauravaram, W. Millan, E. Dawson, and K. Viswanathan. Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction. In *CISP '08*, volume 4058 of *LNCS*, pages 407–420. Springer-Verlag, 2006. 38

[76] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *Journal of the ACM*, 33(4):792–807, 1986. 22, 107

[77] M. Gorski, S. Lucks, and T. Peyrin. Slide Attacks on a Class of Hash Functions. In *Asiacrypt '08*, volume 5350 of *LNCS*, pages 143–160. Springer-Verlag, 2008. 44

[78] S. Halevi and H. Krawczyk. Strengthening Digital Signatures via Randomized Hashing. In *Crypto '06*, volume 4117 of *LNCS*, pages 41–59. Springer-Verlag, 2006. 27, 39, 51

[79] S. Halevi and H. Krawczyk. The RMX Transform and Digital Signagures. In *2nd NIST Hash Workshop*, 2006. 40, 51

[80] S. Hirose. How to Construct Double-Block-Length Hash Functions. In *2nd NIST Cryptographic Hash Workshop*, 2006. 46

[81] S. Hirose, J. H. Park, and A. Yun. A Simple Variant of the Merkle-Damgård Scheme with a Permutation. In *Asiacrypt '08*, volume 4833 of *LNCS*, pages 113–129. Springer-Verlag, 2008. 39, 106

[82] Intel Corp. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2008. 143, 146, 149, 152

[83] Intel Corp. *Intel 64 and IA-32 Architectures Software Developers Manual - Vol. 1 Basic Architecture*, 2008. 145, 146

[84] Intel Corp. *Intel 64 and IA-32 Architectures Software Developers Manual - Vol. 2A Instruction Set Reference A-M*, 2008. 146

[85] Intel Corp. *Intel 64 and IA-32 Architectures Software Developers Manual - Vol. 3B System Programming Guide, Part 2*, 2008. 146

[86] Intel Corp. *Intel 64 and IA-32 Architectures Software Developers Manual - Vol. 2B Instruction Set Reference N-Z*, 2009. 146

[87] Intel Corp. *Intel 64 and IA-32 Architectures Software Developers Manual - Vol. 3A System Programming Guide, Part 1*, 2009. 146

[88] D. Joscak and J. Tuma. Multi-block Collisions in Hash Functions Based on 3C and 3C+ Enhancements of the Merkle-Damgård Construction. In *ICISC '06*, volume 4296 of *LNCS*, pages 257–266. Springer-Verlag, 2006. 38

[89] A. Joux. Multicollisions in Iterated Hash Functions: Application to Cascaded Constructions. In *Crypto '04*, volume 31252 of *LNCS*, pages 306–316. Springer-Verlag, 2004. 35

[90] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Scribner, 1996. (Revised edition). 1

[91] J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In *Eurocrypt '06*, volume 4004 of *LNCS*, pages 183–200. Springer-Verlag, 2006. 16, 36, 37

[92] J. Kelsey and B. Schneier. Second Preimages on n-bit Hash Functions for Much Less than 2^n work. In *Eurocrypt '05*, volume 3494 of *LNCS*, pages 474–490. Springer-Verlag, 2005. 35, 36

[93] J. Kilian and P. Rogaway. How to Protect DES Against Exhaustive Key Search (an analysis of DESX). *Journal of Cryptology*, 14(1):17–35, 2001. 21

[94] V. Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105, 2006. (eprint.iacr.org/2006/105). 13

[95] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley, 2nd edition, 1998. 3

[96] X. Lai and J. Massey. Hash Functions Based on Block Ciphers. In *Eurocrypt '93*, volume 658 of *LNCS*, pages 55–70. Springer-Verlag, 1993. 33

[97] L. Lamport. Password Authentication with Insecure Communication. *Communications of the ACM*, 24(11):770–772, 1981. 1

[98] J. Lee and J. Park. Adaptive Preimage Resistance and Permutation-based Hash Functions. Cryptology ePrint Archive, Report 2009/066, 2009. (eprint.iacr.org/2009/066). 25

[99] W. Lee, D. Chang, S. Lee, S. Sung, and M. Nadi. New Parallel Domain Extenders for UOWHF. In *Asiacrypt '03*, volume 2894 of *LNCS*, pages 208–227. Springer-Verlag, 2003. 44

[100] D. Leurent and P. Nguyen. How Risky is the Random-Oracle Model? In *Crypto '09*, volume 5677 of *LNCS*, pages 445–464. Springer-Verlag, 2009. 71

[101] S. Lucks. A Failure-Friendly Design Principle for Hash Functions. In *Asiacrypt '05*, volume 3788 of *LNCS*, pages 474–494. Springer-Verlag, 2005. 37

[102] M. M. A. Majeed, K. A. Al-Khateeb, M. R. Wahiddin, and M. M. Saeb. Protocol of Secure Key Distribution Using Hash Functions and Quantum Authenticated Channels Key Distribution Process Six-State Quantum Protocol. *Journal of Computer Science*, 6(10):1094–110, 2010. 1

[103] S. Manuel and N. Sendrier. XOR-Hash: A Hash Function Based on XOR. In *WEWRC '07*, 2007. 44

[104] M. Maqableh, A. Samsudin, and M. Alia. New Hash Function Based on Chaos Theory (CHA-1). *International Journal of Computer Science and Network Security*, 8(2):20–27, 2008. 48

[105] S. Mathew and K. P. Jacob. Performance Evaluation of Popular Hash Functions. *World Academy of Science, Engineering and Technology*, 61:449–452, 2010. 140

[106] M. Matsui. Linear Cryptanalysis Method for DEC Cipher. In *Eurocrypt '94*, volume 765 of *LNCS*, pages 386–397. Springer-Verlag, 1994. 51

[107] U. Maurer. Indistinguishability of Random Systems. In *Eurocrypt '02*, volume 2332 of *LNCS*, pages 110–132. Springer-Verlag, 2002. 105, 107, 108

[108] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In *TCC '04*, volume 2951 of *LNCS*, pages 21–39. Springer-Verlag, 2004. 17, 71, 106, 111

[109] U. Maurer and J. Sjödin. Single-key AIL-MACs from any FIL-MAC. In *ICALP '05*, volume 3580 of *LNCS*, pages 472–484. Springer-Verlag, 2005. 31, 42, 43, 55, 107, 114, 115, 116, 117

[110] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, chapter Hash Functions and Data Integrity, pages 321–384. CRC Press, 1996. 15, 46

[111] R. Merkle. One Way Hash Functions and DES. In *Crypto '89*, volume 435 of *LNCS*, pages 428–446. Springer-Verlag, 1989. 15, 19, 32, 33, 54, 70

[112] M. Mihaljevie, Y. Zheng, and H. Imai. A Cellular Automaton Based Fast One-Way Hash Function Suitable for Hardware Implementation. In *PKC '98*, volume 1431 of *LNCS*, pages 217–233. Springer-Verlag, 1998. 48

[113] I. Mironov. Hash Functions: From Merkle-Damgård to Shoup. In *Eurocrypt '01*, volume 2045, pages 166–181. Springer-Verlag, 2001. 120

[114] G. Moore. Cramming More Components onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1965. (Reprinted from Gordon E. Moore, Cramming More Components onto Integrated Circuits, Electronics, pp. 114117, April 19, 1965.). 3

[115] M. Nandi. Toward Optimal Double-Length Hash Functions. In *Indocrypt '05*, volume 3797 of *LNCS*, pages 77–89. Springer-Verlag, 2005. 46

[116] M. Naor and N. Yung. Universal One-Way Hash Functions and their Cryptographic Applications. In *STOC '89*, pages 33–43. ACM, 1989. 26, 44, 120

[117] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) Family, 2007. (volume 72, pages 62212–62220). 30

[118] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*, 2009. (Federal Information Processing Standards Publication 186-3). 1

[119] J. Nielsen. Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-Committing Encryption Case. In *Crypto '02*, volume 2442 of *LNCS*, pages 111–126. Springer-Verlag, 2002. 71

[120] NIST. *The Keyed-Hash Message Authentication Code (HMAC)*, 2002. (FIPS PUB 198). 31

[121] Y. Nito, K. Yoneyama, L. Wang, and K. Ohta. How to Prove the Security of Practical Cryptosystems with Merkle-Damgård Hashing by Adopting Indifferentiability. Cryptology ePrint Archive, Report 2009/040, 2009. (eprint.iacr.org/2009/040). 22

[122] P. Pal and P. Sarkar. PARSHA-256 – A New Parallelizable Hash Function and a Multithreaded Implementation. In *FSE '03*, volume 2887 of *LNCS*, pages 347–361. Springer-Verlag, 2003. 44

[123] J. Patarin. Collisions and Inversions for Damgård's Whole Hash Function. In *Asiacrypt '95*, volume 917 of *LNCS*, pages 305–321. Springer-Verlag, 1995. 47

[124] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholiek Universiteit Leuven, 1993. 29

[125] B. Preneel. Cryptographic Hash Functions. In *3rd Symposium on State and Progress of Research in Cryptography*, pages 161–171, 1993. 47

[126] B. Preneel. The State of Hash Functions and the NIST SHA-3 Competition. In *Inscrypt '09*, volume 5487 of *LNCS*, pages 1–11. Springer-Verlag, 2009. 29

[127] B. Preneel, R. Govaerts, and J. Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In *Crypto '93*, volume 773 of *LNCS*, pages 368–378. Springer-Verlag, 1993. 45

[128] M. Rabin. *Foundations of Secure Computations*, chapter Digitalized Signatures, pages 155–166. Academic Press, 1978. 32, 86

[129] M. Reyhanitabar, W. Susilo, and Y. Mu. Analysis of Property-Preservation Capabilities of the ROX and ESh Hash Domain Extenders. In *ACISP '09*, volume 5594 of *LNCS*, pages 153–170. Springer-Verlag, 2009. 27, 42

[130] M. R. Reyhanitabar, W. Susilo, and Y. Mu. Enhanced Security Notions for Dedicated-Key Hash Functions: Defintions and Relationships. Cryptology ePrint Archive, Report 2010/022, 2010. (eprint.iacr.org/2010/022). 27

[131] T. Ristenpart and T. Shrimpton. How to Build a Hash Function from any Collision-Resistant Function. In *Asiacrypt '07*, volume 4833 of *LNCS*, pages 147–163. Springer-Verlag, 2007. 22

[132] R. Rivest. Abelian Square-Free Dithering for Iterative Hash Functions. In *1st NIST Cryptographic Hash Workshop*, 2005. 41

[133] R. Rivest, B. Agre, D. Bailey, C. Crutchfield, Y. Dodis, K. E. Fleming, A. Khan, J. Krishnamurthy, Y. Lin, L. Reyzin, E. Shen, J. Sukha, D. Sutherland, E. Tromer, and Y. L. yin. *The MD6 Hash Function: a Proposal to NIST for SHA-3*, 2008. (groups.csail.mit.edu/cis/md6). 45, 51, 54, 55, 122

[134] P. Rogaway. Formalizing Human Ignorance: Collision-Resistant Hashing Without the Keys. In *Vietcrypt '06*, volume 4341 of *LNCS*, pages 211–228. Springer-Verlag, 2006. 14, 25, 50, 69

[135] P. Rogaway and T. Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In *FSE '04*, volume 3017 of *LNCS*, pages 371–388. Springer-Verlag, 2004. 14, 15, 25, 42, 61, 120, 121

[136] P. Rogaway and J. Steinberger. Constructing Cryptographic Hash Functions from Fixed-Key Blockciphers. In *Crypto '08*, volume 5157 of *LNCS*, pages 433–450. Springer-Verlag, 2008. 46

[137] S. Ross. *A First Course in Probability*. Pearson Education Ltd., 2008. 6

[138] M.-J. Saarinen. Linearization Attacks Against Syndrome Based Hashes. In *Indocrypt '07*, volume 4859 of *LNCS*, pages 1–9. Springer-Verlag, 2007. 47

[139] P. Sarkar. Masking Based Domain Extenders for UOWHFs: Bounds and Constructions. In *Asiacrypt '04*, volume 3329 of *LNCS*, pages 187–200. Springer-Verlag, 2004. 44

[140] P. Sarkar and P. Scellenberg. A Parallel Algorithm for Extending Cryptographic Hash Functions. In *Indocrypt '01*, volume 2247 of *LNCS*, pages 40–49. Springer-Verlag, 2001. 44

[141] V. Shoup. A Composition Theorem for Universal One-Way Hash Functions. In *Eurocrypt '00*, volume 1807 of *LNCS*, pages 445–452. Springer-Verlag, 2000. 42, 120

[142] T. Shrimpton and M. Stam. Building a Collision-Resistant Compression Function from Non-compressing Primitives. In *ICALP '08*, volume 5126 of *LNCS*, pages 643–654. Springer-Verlag, 2008. 46, 122

[143] M. Stam. Beyond Uniformity: Better Security/Efficiency Tradeoffs for Compression Functions. In *Crypto '08*, volume 5157 of *LNCS*, pages 397–412. Springer-Verlag, 2008. 46

[144] M. Stam. Blockcipher-Based Hashing Revisited. In *FSE '09*, volume 5665 of *LNCS*, pages 69–83. Springer-Verlag, 2009. 46

[145] M. Stevens, A. Lenstra, and B. Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In *Eurocrypt '07*, volume 4515 of *LNCS*, pages 1–22. Springer-Verlag, 2007. 7

[146] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, and D. Molnar. Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certification. In *Crypto '09*, volume 5677 of *LNCS*, pages 55–69. Springer-Verlag, 2009. 7, 13

[147] D. Stinson. Some Observations on the Theory of Cryptographic Hash Functions. *Designs, Codes and Cryptography*, 38(2):259–277, 2006. 14

[148] D. Stinson and J. Upadhyay. On the Complexity of the Herding Attack and Some Related Attacks on Hash Functions. Cryptology ePrint Archive, Report 2010/030, 2010. (`eprint.iacr.org/2010/030`). 37

[149] J.-P. Tillich and G. Zemor. Collisions for the LPS Expander Graph Hash Function. In *Eurocrypt '08*, volume 4965 of *LNCS*, pages 254–269. Springer-Verlag, 2008. 47

[150] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely. High-Speed Hardware Implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein. Cryptology ePrint Archive, Report 2009/510, 2009. (eprint.iacr.org/2009/510). 142

[151] G. Tsudik. Message Authentication with One-Way Hash Functions. *ACM SIGCOMM Computer Communication Review*, 22:29–38, 1992. 34

[152] P. C. van Oorschot and M. J. Wiener. Parallel Collision Search with application to Hash Functions and Discrete Logarithms. In *ACM CCS '94*, pages 210–218, 1994. 7

[153] J. von Neumann. *The World of Physics: A Small Library of the Literature of Physics from Antiquity to the Present*, chapter The General and Logical Theory of Automata, pages 606–607. Simon and Schuster, New York, 1987. (Originally presented at the Hixon Symposium on September 20, 1948, at the California Institute of Technology). 48

[154] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/1999, 2004. (eprint.iacr.org/2004/199). 2, 30, 49, 69

[155] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In *Crypto '05*, volume 3621 of *LNCS*, pages 17–36. Springer-Verlag, 2005. 2, 30, 31, 33, 49, 69

[156] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Eurocrypt '05*, volume 3494 of *LNCS*, pages 19–35. Springer-Verlag, 2005. 2, 30, 31, 33, 49, 69

[157] C. Wenzel-Benner and J. Gräf. XBX: eXternal Benchmarking eXtension for the SUPERCOP Crypto Benchmarking Framework. In *CHES '10*, volume 6225 of *LNCS*, pages 294–305. Springer-Verlag, 2010. 142

[158] S. Wolfram. *A New Kind of Science*. Wolfram Media, 2002. 48

[159] K. Yasuda. Boosting Merkle-Damgrd Hashing for Message Authentication. In *Asiacrypt '07*, volume 4833 of *LNCS*, pages 216–231. Springer-Verlag, 2007. 107

[160] K. Yasuda. How to Fill Up Merkle-Damgård Hash Function. In *Asiacrypt '08*, volume 5350 of *LNCS*, pages 272–289. Springer-Verlag, 2008. 15, 27

[161] K. Yasuda. A Double-Piped Mode of Operation for MACs, PRFs and PROs: Security beyond the Birthday Barrier. In *Eurocryp '09*, volume 5479 of *LNCS*, pages 242–259. Springer-Verlag, 2009. 38, 112

[162] X. Yi. Hash Function Based on Chaotic Tent Maps. *IEEE Transactions on Express Briefs*, 52(6):354–357, 2005. 48

[163] K. Yoneyama, S. Miyagawa, and K. Ohta. Leaky Random Oracle (Extended Abstract). In *ProvSec '08*, volume 5324 of *LNCS*, pages 226–240. Springer-Verlag, 2008. 22

[164] G. Yuval. How to Swindle Rabin. *Cryptologia*, 3(3):187–191, 1979. 11

Appendix A

Engineering Aspects of Hash Functions

Hash functions have numerous applications in cryptography, from public key to cryptographic protocols and cryptosystems. However, while substantial effort was invested on designing "secure" hash functions, other engineering aspects that may affect their use in practice were inadvertently overlooked. In this appendix, we argue that in some applications, the efficiency of hash functions is as important as their security. Unlike most of the existing related works in the literature (which merely report on efficiency figures of some popular hash functions without discussing how and why these results were obtained), we not only discuss how to carry out efficiency evaluations, but we also provide a set of optimisation guidelines to assist implementers in optimising their implementations. We demonstrate this by adopting an existing SHA-1/SHA-2 implementation and show how minor optimisation can lead to significant efficiency improvements. The contents in this appendix was published in [2].

A.1 Introduction

Today, cryptographic hash functions play a major role in most cryptographic applications. Abstractly, hash functions are transformation procedures that given data, they return (small, fixed) fingerprints. A typical hash function consists of two components: a compression function and a construction. The compression function is a function mapping a larger fixed-size input to a smaller fixed-size output, and the construction is the way the compression function is being repeatedly called to process a variable-length

message in a finite fixed-length blocks. Most of the literature is exclusively concerned with the design and cryptanalysis of hash functions. However, while the security of hash functions is certainly a highly important aspect, for some applications, especially the ones processing large amount of data, the efficiency (how fast the hash function is) is also important. Although there have been some efforts in evaluating the performance of hash functions, e.g. [105], it is clear that this is a largely overlooked evaluation criterion. Even the contributions that provide such efficiency evaluations, they generally only make the efficiency reports, without elaborating on how to improve[1] them. In this appendix, we try to do this by considering implementations targeted for Intel platforms.

Appendix outline. This appendix is organised as follows. In section A.2, we discuss the main factors affecting the efficiency of hash functions (and any code in general). Section A.3 provides a concise overview of contemporary Intel platforms and some of their advanced architectural features. Our main discussion is in section A.4 where we investigate how to optimise code on Intel platforms; though most of these optimisation techniques are generic and applicable to other platforms. In section A.5 we show how to carry out performance evaluations of hash functions and present a sample SHA-1/SHA-2 optimisation case study in which we demonstrate how minor optimisations can greatly improve the overall efficiency of hash functions.

A.2 Efficiency Evaluation

The efficiency of any cryptographic primitive can significantly influence its popularity. For example, Serpent [8] was one of the AES (Advanced Encryption Standard) competition finalists and it was described by NIST (the competition organiser) as having a *high* security margin. However, in the last round of the competition, Serpent failed in favour of Rijndael [53], which was described as having just an *adequate* security, because Serpent was very slow in software compared to Rijndael. In this appendix, we will be mainly concerned with the software efficiency of hash functions (but see section A.2.2 for a brief discussion about hardware efficiency).

A.2.1 Software Optimisation

Generally, there are two types of software optimisations, *high-level* and *low-level* optimisations. In high-level optimisation, a cross-platform implementation written in a high-level language, such as C, is optimised. However, different compilers may treat

[1] We note that some SHA-3 submissions include optimisation discussions, e.g. [69], but these by no means are comprehensive.

high-level code slightly differently such that a code might be considered optimised only if it was compiled by a particular compiler. On the other hand, low-level optimisation involves optimising a machine (or assembly) code, and is rarely cross-platforms since different platforms often use different instruction sets. While optimising a low-level code is tedious and error-prone, it gives the highest degree of control over the code. In general, efficiency requirements highly depend on the application, and thus the targeted application should also be taken into account when implementing a hash function. Software efficiency of hash functions can be influenced by several factors including, the platform in which the hash function is executed, the compiler used to compile the hash function code, and the executing operating system (hash functions optimised for 64-bit operating systems are slower in 32-bit operating systems, e.g., Skein [69]).

Platforms. Both high-level and low-level optimisations are usually *tuned* for a specific platform. For example, in the SHA-3 competition[2] the reference platform in which the candidates were instructed to evaluated their submissions on was Intel Core 2 Duo, thus most of the candidate submissions were especially tuned to be optimal in Intel platforms (which mean that they may not be optimal in other platforms!). Platforms can be roughly classified as follows:

- *High-end.* These are platforms with high computational and memory resources, and usually based on 32-bit or 64-bit architectures, often with multiple processing cores. Examples of such platforms include Intel and AMD.
- *Intermediate.* These are most 16-bit and 32-bit microcontrollers with moderate computational resources (Note that some microcontrollers are low-end platforms). Examples of such platforms include ARM and AVR.
- *Low-end.* These are 8-bit platforms with limited computational and memory (usually kilobytes) resources. Examples include Smart Cards and FRID.

Compilers. Another very important factors to consider when investigating hash functions efficiency is the sophistication of the compiler. Most of the available compilers (commercial and open source) like Microsoft Visual Studio and GCC are sophisticated enough to automatically optimise the code. However, these compilers sometimes apply some optimisation techniques that may not be optimal for all platforms. Thus, it is advisable to compile the code with several different compilers and use different optimisation switches, then only choose the optimum one for a target platform, though this

[2]For comprehensive resource about SHA-3 competition and all its candidates, see http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

process may be tedious. One would think that a commercialised compiler developed by the vendor of a particular platform would outperform other open source or third-party commercial compilers. However, Wenzel-Benner and Graf [157] showed that this is not always the case when they implemented several SHA-3 hash function candidates on an ARM Cortex platform and then compiled them twice, once by ARM-CC compiler (ARM's own compiler) and another with GCC compiler (open source compiler). Surprisingly, they found that in some cases, GCC compiled code is more efficient than that compiled by ARM-CC.

Instruction Sets. High-level code will eventually be converted into a low-level (machine) code that consists of instructions. A platform with only several tens of instructions will most likely not perform as well as another with hundreds of instructions, basically because no matter what optimisation techniques are applied, if no efficient instructions exist for a particular operation, the code will need to be converted to a series of instructions implementing that operation. Most recent platforms adopt the so-called Single Instruction Multiple Data (SIMD) technology, which provides instructions allowing for parallel data execution (see section A.3.1).

A.2.2 Hardware Optimisation

Although hardware implementation and optimisation is not our main focus here, in this section we discuss a few interesting results of recent hardware evaluation of SHA-3 candidates. In [150], Tillich *et al.* presented optimised hardware implementations of the 14 SHA-3 round 2 candidates. Their results show that Keccek and Luffa significantly outperform all other candidates. The authors did not make any conclusions, but we point out that Keccek and Luffa are the only round 2 candidates adopting permutation-based (sponge and sponge-like) constructions. Although not strictly a hardware implementation aspect, Intel has recently released a new instruction set named AES-NI [30]. Hash functions based on AES, such as LANE, ECHO and Lesamnta, will benefit from these instructions significantly. However, in order for a hash function to make the most of these new AES-NI instructions, it should be based on an unmodified AES construction.

A.3 Intel Platform

The scope of this appendix is restricted to software optimisation on Intel platforms because these appear to be the most common platforms nowadays. Currently, the most popular Intel processors are those of families descending from the Core architecture

which introduced many revolutionary features to improve the processing performance, these features include:

- *Wide Dynamic Execution.* With Core Microarchitecture, each core can execute up to 4 instructions simultaneously. Intel also introduced Macro-fusion and Micro-fusion, which fuse micro-ops.

  - Micro-fusion operates on low-level processor instructions called micro-operations[3] (micro-ops) to fuse multiple micro-ops of the same instruction into a single (complex) micro-op.
  - Macro-fusion operates on instructions and combines common pairs into a single micro-op. In Intel Core Microarchitecture macro-fusion is only supported in 32-bit mode but with the introduction of Intel Microarchitecture (Nehalem), macro-fusion is now supported in the 64-bit mode too. Currently, the first instruction of the marco-fused pair is restricted to CMP (compare) or TEST (test) instructions followed by a conditional jump instruction that checks the CF and/or ZF flags in the EFLAG register; see section A.3.2. In Intel Core Microarchitecture, supported conditional jump instructions are: JA/JNBE, JE/JZ, JNA/JBE, JNE/JNZ, JAE/JNB/JC and JNAE/JB/JC — condition codes supporting unsigned values. Intel Microarchitecture (Nehalem) adds support to: JL/JNGE, JGE/JNL, JLE/JNG and JG/JNLE — condition codes supporting signed values [82]; see table A.1 for description of condition codes.

- *Advanced Smart Cache.* This feature allows each core to dynamically utilise up to 100% of the (fast) L2 cache if available, which was previously not possible resulting in an inefficient use of the cache.

- *Advanced Digital Media Boost.* This feature improved the execution of SIMD instructions by allowing the whole 128-bit instruction to be executed in one clock cycle, which used to utilise several clock cycles previously.

A.3.1 SIMD Instruction Set

In SIMD (Single Instruction, Multiple Data), a single instruction operates on multiple data simultaneously achieving a data level parallelism. SMID instructions are especially useful when the same operation needs to be executed repeatedly on different data (e.g.

[3] Note that micro-ops are executed directly by the hardware and should not be confused with normal instructions which are themselves composed of several micro-ops

hashing a message). In 1997, Intel introduced the MMX instruction set based on SIMD, which was later improved by introducing the SSE (Streaming SIMD Extensions). Later versions of SSE include: SSE2, SSE3, SSSE3, SSE4, SSE5 and recently AVX. Most of these instructions were introduced to support heavy processing applications like multimedia, gaming, signal processing and modelling. Today, SIMD instructions are available in most of the recent platforms from Intel and AMD to ARM.

A.3.2 Registers in Intel Platforms

Registers are very fast storage mediums located near the processors. Below we provide brief descriptions of some register types found in most Intel platforms:

- General-purpose Registers: these are eight 32-bit registers (EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP) hold operands and memory pointers. In 64-bit mode, there are sixteen 64-bit registers; these are EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP, R8D-R15D for 32-bit operands, and RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8-R15 for 64-bit operands.
- *Segment Registers*: these are six 16-bit registers (CS, DS, SS, ES, FS, GS) used to hold pointers.
- *MMX Registers*: these are eight 64-bit registers (MM0 – MM7) introduced with the MMX technology to perform operations on 64-bit data.
- *XMM Registers*: these are either eight (in 32-bit mode) or sixteen (in 64-bit mode) 128-bit registers (XMM0 – XMM15), introduced to handle the SSE 128-bit data types.
- *MXCSR Register.* This is a 32-bit register used to control some of the SSEx floating-point operations.
- *EIP Register.* This is a 32-bit register that holds the instruction pointer which points to the next instruction to be executed.
- *EFLAG Register*: this is a single 32-bit (in 32-bit mode) register used to reflect the results of comparison, arithmetic and other instructions by setting its flags appropriately. In 64-bit mode, EFLAG is called RFLAG and is 64-bit, the lower 32-bit are identical to EFLGA, and the upper 32-bit are reserved.

The EFLAG register contains 1 control flag, 6 status flags, 11 system flags and the rest are reserved bits. Below we elaborate on EFLAG's 6 status flags, which are relevant to the discussions in the following sections.

Code	Description	Flag
O	Overflow	OF=1
NO	No Overflow	OF=1
B/NAE	Below/Not Above or Equal	CF = 1
NB/AE	Not Below/Above or Equal	CF = 0
E/Z	Equal/Zero	ZF = 1
NE/NZ	Not Equal/Not Zero	ZF = 0
BE/NA	Below or Equation/Not Above	CF $\vee$ ZF = 1
NBE/A	Not Below or Equal Above	CF $\vee$ ZF = 0
S	Sign	SF = 1
NS	No Sign	SF = 0
P/PE	Parity/Parity Even	PF =1
NP/PO	Not Parity/Parity Odd	PF = 0
L/NGE	Less/Not Greater than or Equal	SP $\oplus$ OF = 1
NL/GE	Not Less than/Greater than or Equal	SF $\oplus$ OF = 0
LE/NG	Less than or Equal/Not Greater than	((SF $\oplus$ OF) $\vee$ ZF) = 1
NLE/G	Not Less than or Equal/Greater than	((SF $\oplus$ OF) $\vee$ ZF) = 0

Table A.1: Intel condition codes

- *Carry Flag* `CF`: set if a carry or borrow of an arithmetic operation is generated.
- *Parity Flag* `PF`: set if the number of 1's in the least significant byte of the result from the previous operation is even (indicating even parity), otherwise it is unset (indicating odd parity).
- *Adjust Flag* `AF`: this flag is primarily used in BCD (binary-coded decimal) arithmetic and is set only if the previous operation generates a carry or borrow.
- *Zero Flag* `ZF`: set if the result of the previous operation is zero.
- *Sign Flag* `SF`: used with signed values and is set to the same value of the sign bit (most significant bit), which is 0 if positive or 1 if negative.
- *Overflow Flag* `OF`: set if the result of an operation doesnt fit the specified destination operand (e.g. storing a 64-bit value in a 32-bit register).

These flags can be tested by other instructions by suffixing a "condition code" to the instruction. Some condition codes are described in table A.1 [83].

As indicated in table A.1, some condition code mnemonics are synonym to others, like B (Below) and NAE (Not Above or Equal). Intel adopts this convention for situations where using one mnemonic is more intelligible than the other. Also, When dealing with unsigned values, the below/above mnemonics are used; but when dealing with signed values, the greater than/less than mnemonics are used instead.

A.4 Intel Optimisation

Compilers, like the Intel C++ compiler, will certainly try to optimise the code, but they might sometimes make wrong decisions. To take advantage of the available powerful instructions, it is desirable to code directly in assembly where we have full control over the flow of the program. However, as discussed earlier, programming in assembly is tedious, time consuming and error-prone. Therefore, it will only worth coding the most critical and frequently called parts of the program in assembly. These critical parts may be a particular function that is being called very frequently, or a loop iterating many times. For example, the performance of a hash function may significantly improve if its compression function (which is repeatedly called) is coded in assembly. Such critical parts can be spotted by running a performance analyser/profiler software, like Intel's VTune.

When optimising an implementation for a specific platform, not only it may not be optimal for other platforms, but it also may not be optimal for different families from the same platform. That is, a particular set of optimisation techniques targeted for Pentium processors, for example, may not be optimal for Core or Core 2 processors. Nevertheless, when considering Intel platforms, we believe that developing a generic code optimised for the Intel Core Microarchitecture is, in general, likely to be an optimal solution for most current and near future processors but may not be so for older processor generations. Another solution would be to explicitly write multiple versions of a particular code, each optimised for a different processor, then, at execution time, the compiler uses the `CPUID` instruction to identify the platform it is running on and only compiles the appropriate version of the code. This approach is, however, not practical if the code size has to be restricted (e.g., targeting low-end platforms such as RFID or smart cards).

In the following sections we discuss a few optimisation techniques [82, 83, 84, 86, 87, 85] developed especially to optimise code targeting Intel platforms and applicable to most Intel Microarchitecture including Intel Core Microarchitecture, Enhanced Intel Core Microarchitecture and Intel Microarchitecture (Nehalem); but note that some of these techniques are generic and may also be applicable to other platforms.

A.4.1 Instruction Selection

Most of the unexpected results/errors are caused by poor selection of instructions. Choosing which instructions to use in a program may not only affect the execution of the program, but also its efficiency. In this section we briefly discuss a few considerations when selecting instructions in an Intel's platform.

Micro-operations. Each instruction is decoded into micro-operations (micro-ops) before it is executed. Intuitively, an instruction that decodes to less micro-ops runs faster. Thus, since complex instructions are usually being decoded to more micro-ops than those decoded from other simpler instructions, it is recommended to use a series of simple instructions that decode to fewer micro-ops, than using a single complex instruction decoding to more micro-ops.

`INC/DEC` **instructions.** In Intel, it is better to use the `ADD` (addition) and `SUB` (subtraction) instructions instead of `INC` (increment) and `DEC` (decrement) instructions because `INC/DEC` only update a subset of the flags in the EFLAG register. This is especially problematic when dealing with condition codes, which are dependant on the EFLAG register's flags. On the other hand, `ADD/SUB` instructions first clear all the flags in the EFLGA register before updating the appropriate ones based on the addition/subtraction result, so if any of the flags was previously set by an earlier instruction, it will be reset. This is not the case with `INC/DEC` since they only update the flags affected by the increment/decrement result and ignore the rest of the flag, which can create false dependency one previous instructions.

Shift/Rotate Instructions. Shift instructions are less expensive than rotate instructions. However, rotate by 1 has the same overhead as the shift instructions. Hence, it might be more efficient to use a series of rotate by 1 instruction for small number of rotations.

`CMP/TEST` **Instructions.** `TEST` instruction ANDs its operands and updates the EFLAG register. If the result of the AND operation is not needed, then using `TEST` is better than using the `AND` instruction because `AND` wastes extra cycles to produce the result. `TEST` incurs less overhead than the `CMP` instruction and is preferred where possible. For example, using the `TEST` instruction on a register with itself is the same as comparing the register to zero. It is also possible to compare values with zero by condition codes if the appropriate flags in EFLGA register were set by earlier instructions, in which case neither `TEST` nor `CMP` is needed.

A.4.2 Optimising Branches

Branches (also called jumps) can greatly influence the performance of the program. Branches are points in the program where the flow of the program is interrupted and diverted. This jump instruction from a point in the program to another certainly incurs

considerable processing overhead. Branches present in assembly code and are comparable to the condition statements ("if" statements) in high-level languages. Branches are usually conditional (based on conditions), but they can sometimes be unconditional where the flow of the program always jumps to where the branch points to as soon as it reaches the branch; if the branch is executed, we say the branch is taken, otherwise, it is not taken, a technique called *branch prediction* predicts whether a branch is more likely to be taken or not taken. It is important to predict a branch especially in pipelined processors (these are almost all the modern processors) because when pipelining instructions, the address of the following instruction has to be known before the execution of the current one; that is, if a branch is to be taken, the following address is going to be the address of the first instruction at the portion of the code where the branch jumps to, otherwise, the following address is the address of the next sequential instruction after the branch instruction. However, even with efficient branch prediction mechanism, it is still advisable to minimise branches as much as possible because even if a branch has been correctly predicted, there is still an overhead for actually taking the branch. Below we discuss some guidelines for optimising branches (though, we do not explicitly discuss any branch prediction algorithm).

It is always recommended that condition codes be used instead of branches where possible. condition codes are dependent on the EFLAG register and are usually proceeded by `CMP` or `TEST` instructions which set the flags in EFLAG appropriately. In particular, the instructions `CMOVcc` or `SETcc` (where `cc` is a condition code) are preferred over branches (note that `SETcc` can only set operands to 1 or 0; if different values are required, `CMOVcc` is used). For example, consider the following C code:

```
if (x >= y) {x = 1;} else {x = 0;}
```

In assembly, this can be written as:

```
CMP eax, ebx    ;compare values of eax and ebx
Jge Gtr         ;if eax >= ebx, jump to 'Gtr'
MOV eax 0       ;otherwise, set eax = 0
JMP Less        ;jump to 'Less' where the
                ;rest of the program is
Gtr:
Mov eax 1       ;set eax = 1
Less:
...
```

Compare this with the following optimised code:

```
CMP    eax, ebx   ;compare  eax and ebx
CMOVge eax, #1    ;if eax >= ebx, eax = 1
CMOVl  eax, #0    ;otherwise, eax = 0
```

A.4.3 Optimising Loops

With the advent of Core Microarchitecture, Intel introduced the Loop Stream Detector (LSD) which expedites the execution of loops containing up to 18 instructions. When a loop is detected, the loop instructions are stored in a special LSD buffer and the fetch and branch prediction stages are powered off until the loop is completed. Intel Microarchitecture (Nehalem) further improved LSD by moving it beyond the decode stage to hold up to 28 micro-ops for immediate execution, powering off all the pipeline stages except the execute stage; here, the LSD buffer is similar to the trace cache[4]. Even with the presence of LSD, the implementer can still further optimise loops. In the following sub sections we discuss a few generic loop optimisation techniques that can be tuned for other platforms (not just Intel); we will either demonstrate these techniques using high-level language examples, or Intel instructions if a particular technique is intrinsic to Intel.

Loop unrolling. Unrolling a loop entails reducing the number of loop iterations by increasing the size of the loop body. Since each loop incurs extra overhead for checking the end-of-loop condition at the end of each iteration, minimising the number of iterations does greatly optimise its execution. Unrolling a loop, however, increases the size of the code and may congest the trace cache. Thus, it is recommended to only unroll the frequently called loops. Intel recommends that a loop should not be iterated more than 16 times, and if it does, it should be unrolled to keep this maximum number of iterations [82]. For example, suppose that the operations `Operation1` and `Operation2` need to be executed 8 times each, this can be written as:

```
for (i = 0; i < 8; i++)
   {Operation1; Operation2;}
```

with loop unrolling, the above code can be re-written more efficiently as:

```
for (i = 0; I < 4; i++){
   Operation1; Operation2;
   Operation1; Operation2;}
```

[4]Trace cache is part of the first-level cache and is used to hold the decoded instructions before execution.

This potentially saves four iterations and consequently four end-of-the-loop condition checks. For loops that iterate many times, saving the end-of-loop condition does significantly improve the speed of the execution.

Loop-blocking. A very effective technique for optimising loops is loop-blocking. This technique is useful when dealing with large amount of data. If the data on which the loop is operating is large, the cache might not be sufficient to hold the data during the whole execution time; then it stores it in memory, which has a slow access time. In this case, using loop blocking allows for partitioning the loop into smaller chunks to operate on data with size small enough to fit in the cache. These chunks are then executed in turn by reusing the cache every time a new chunk is executed. Consider the following example with a cache of size 8 bytes (the cache can store 8 bytes at any given time, any extra values are stored in memory):

```
for (i = 0; i <= 16; i++){
     Function1(x[i]);}
for (i = 0; i <= 16; i++){
     Function2(x[i]);}
```

Since the cache can only store 8 bytes, when executing `Function1` only 8 bytes of array `x` can be stored in the cache, the other 8 bytes will be stored in memory and will need to be loaded for `Function2`. Compare this with the following code after applying loop-blocking:

```
for (i = 0; i < 2; i+=8) {
   for (j = i; j < min(16,8+i); j++){
      Function1(x[j]);}
   for (j = i; j < min(16,8+i); j++){
      Function(x[j]);}}
```

Here, instead of operating on the 16 bytes of the array `x` at once, the loop is divided into two portions of size 8 each. `Function1` operates on the first 8 bytes of array `x` which fit in the cache, then the same 8 bytes are transferred to `Function2` to be operated on. Once both `Function1` and `Function2` finish executing the first 8 bytes of array `x`, the cache is purged for the other 8 bytes of array `x` and the same process is repeated.

Decrementing Loop. When writing a loop, it is more efficient to use a decrementing loop rather than an incrementing one. An incrementing loop requires 3 instructions: an addition instruction `ADD` to increment the loop counter, a compare instruction `CMP`

to compare the loop counter to the maximum value at which the loop should terminate and, a conditional branch `JLE` to iterate through the loop. On the other hand, a decrementing loop will only need two instructions: a subtraction instruction `SUB` to decrement the loop counter, and a conditional branch instruction `JNZ`; a comparison instruction is not needed in this case since `SUB` will set the condition flag `ZF` in the EFLAG register when it reaches zero (end of loop) which is then tested by the conditional branch instruction `JNZ` before iterating through the loop and would only loop if `ZF` is not set. For example, the program below implements a loop that calculates 5! (5 factorial).

```
for (i = 1; i <= 5; i++){
     factorial *= i; }
```

which will be translated into assembly as follows:

```
MOV eax, #1          ;eax=1
MOV ebx, #1          ;ebx=1
Loop:
MUL eax, eax, ebx    ;eax = eax * ebx
ADD ebx, ebx, #1     ;ebx++
CMP ebx, #0x05       ;ebx = 5 ?
JLE Loop             ;branch if ebx <= 5
```

but, if we rewrite it using a decrementing loop:

```
for (i = 5; i >=1; i--){ factorial *=i; }
```

the assembly will be translated as follows:

```
MOV eax, #1          ;eax=1
MOV ebx, #1          ;ebx=1
Loop:
MUL eax, eax, ebx    ;eax = eax * ebx
SUB ebx, ebx, #1     ;ebx--
JNZ Loop             ;branch if ZF != 0
```

saving one instruction (one cycle) per iteration.

A.4.4 Optimising Functions

If a particular function is frequently called, it is recommended to inline it. Inlining a function involves creating a local copy of the function inside the calling program and thereby eliminating the overhead of jumping outside the program and back again repeatedly while calling it. Like loop unrolling, inlining a function increases the code size; hence, it is advisable that only small functions are inlined for an implementation targeting a memory-constrained platforms. For example, to optimise a hash function, it would worth inlining only its compression function or small parts of the compression function if they are being heavily used during the hashing process.

Moreover, the decision of whether *calling* a function or *jumping* to it can affect the efficiency. When calling a function, more overhead is incurred because it requires a return and the return address should be saved in the stack. On the other hand, when jumping to a function, neither is required. Therefore, if a return from a function is not necessary, it is recommended to jump to that function rather than calling it.

A.4.5 Optimisation for SIMD

The following techniques apply to processors supporting MMX and SSE instruction sets; these are (mostly) processors based on Intel Core Microarchitecture, Enhanced Intel Core Microarchitecture and Intel Microarchitecture (Nehalem) as well as a few Pentium processors. For the code to benefit from the SSE instructions, it has to be *vectorised*. Vectorisation is the process of parallelising the code to take advantage of the inherent parallelism of SSE instructions. For example, four 32-bit double words[5] can be stored in a single 128-bit XMM registers for a single SSE instruction to operate on simultaneously.

There are several ways in which code can be optimised for MMX/SSE technologies. The most straightforward way is to code directly in assembly. This can be either a standalone assembly implementation or an assembly embedded in a C/C++ code using the inlined assembly extension to C/C++. Another way is to use intrinsic functions, which can directly access the SSE instructions using sufficiently large data types (e.g. __m128 which is 128-bit long). These functions are built-in to the compiler and will be inlined at compilation time. However, intrinsic functions are compiler-dependant and not portable. Alternatively, special classes [82] implemented for this purpose can also be used, but these are, again, compiler-dependant.

Another important consideration when dealing with MMX/SSE instructions (or

[5] In XMM registers, a byte is 8-bits, a word is 16-bits, a double word is 32-bit, a quadword is 64-bit and a double quadword is 128-bit, that is, a single XMM register consists of either 16 bytes, 8 words, 4 double words, 2 quadwords or 1 double quadword.

any instruction set in general) is data alignments. It is important to align the MMX operands to 64-bit boundaries to fit on the 64-bit MMX registers because it is very expensive to operate on unaligned data. Similarly, SSE operands should be aligned to 128-bit boundaries to fit on the 128-bit XMM registers. One way to align unaligned data is to pad the operands appropriately. Also, in some cases, rearranging the data may help in aligning it. Carefully rearranging data of different sizes (and types) assigned to structures is an example of such practice.

A.5 Performance Evaluation

Instructions are executed in clock cycles, which are the fundamental units of the CPU clock rate. CPU's that operate on 2.0 GHz clock rate, for example, execute 2×10^9 instructions per second. Therefore, the performance of a particular code is usually evaluated by counting the clock cycles the CPU wastes in executing it; the fewer these cycles, the more efficient the code. In this section, we briefly describe the most common ways of counting CPU cycles in Intel platforms. When evaluating the efficiency of hash functions, we usually count cycles/byte. In order to calculate how many cycles each byte of the message takes to be hashed, the overall number of clock cycles for hashing the whole message is first counted, then divided by the message size in bytes.

To count the clock cycle in Intel platforms, the `RDTSC` (Read Time Stamp Counter) instruction is used. `RDTSC` indicates how many clock cycles the CPU has executed since it was powered up or reset. There are a number of header files that can be used to inline the `RDTSC` instruction, we adopt cycle.h[6], which uses the function `getticks()` to record cycle readings and then the function `elapsed(.,.)` to subtract readings and get the actual number of cycles wasted during execution.

```
#include "rdtsc.h"
...
tsc_counter t1,t2,t3;
t1 = getticks();     //take first reading
   run hash function ...
t2 = getticks();    //take second reading
t3 = elapsed(t2,t1); //get the difference
```

However, when executing the code, the CPU is not exclusively reserved for this executing process; instead, in reality, it usually executes other processes, such as OS transactions etc. in parallel. Consequently, if we count the clock cycles as above only

[6]available from: www.fftw.org/cycle.h (accessed March 2011).

once, it is very likely that the obtained result will also include other delays not caused by the code under evaluation. There are two common ways to improve the accuracy of the clock cycle counting process: average count or least count.

A.5.1 Average Count

One way to count the clock cycles is to execute the code (i.e. hash function) several times and accumulate the clock cycle readings of all executions, then the accumulated clock cycle count is averaged. The pseudocode below illustrates this procedure.

```
tsc_counter t1,t2,t3;
for (x=0; x < MAX; x++) {
t1 = getticks();
   run hash function ...
t2 = getticks();
t3 += elapsed(t2 - t1)    //accumulate
} t4 = t3 / MAX; //average
```

A.5.2 Least Count

Another way is to execute the code several times and only consider the smallest cycle count. However, in this case there is a risk of caching. When the code is first executed, it is most likely that it will be executed from the memory, but after a few iterations, the CPU may choose to move the code to the cache and execute it from there. The pseudocode below illustrates this procedure.

```
tsc_counter t1, t2, t3, t4 = -1;
for (x=0; x < MAX; x++) {
t1 = getticks()();
   run hash function ...
t2 = getticks()();
t3 = elapsed(t2 - t1);
t4 = (t4 > t3 ? t3 : t4); }
```

A.5.3 Demonstration

To demonstrate how optimising the code can affect the overall efficiency of the implementation, we adopted Brian Gladman's implementations[7] of SHA1 and SHA2 and

[7] Available from: www.gladman.me.uk (accessed March 2011).

very slightly optimised them, then compared our optimised implementations to Gladman's original ones. The experiments were carried out on Intel Core 2 Duo 3.00 GHz, 4.00 GB of RAM, running 32-bit Windows Vista Home Premium, and using GNU GCC compiler, while hashing messages with sizes ranging from 100 to 1000,000 Bytes. Figure A-1 plots the results. As shown in the figure, apart from some discrepancies in SHA-384 and SHA-512 implementations for short messages, we observe a general efficiency improvement, especially in SHA-224 and SHA-256 since these handle data in 32-bit words, which suits our testbed OS. We also observe that the hashing rate is higher in large messages since the processing overhead of the message initialisation and hashing finalisation fades out. In this experiment, we deliberately only made minor optimisation on the loops (without unrolling them and so maintaining the original code size) to observe what effect minor optimisation can have on the efficiency of the code. We expect even higher efficiency gain should more optimisation techniques are applied, but that may trade off code size and coding effort.

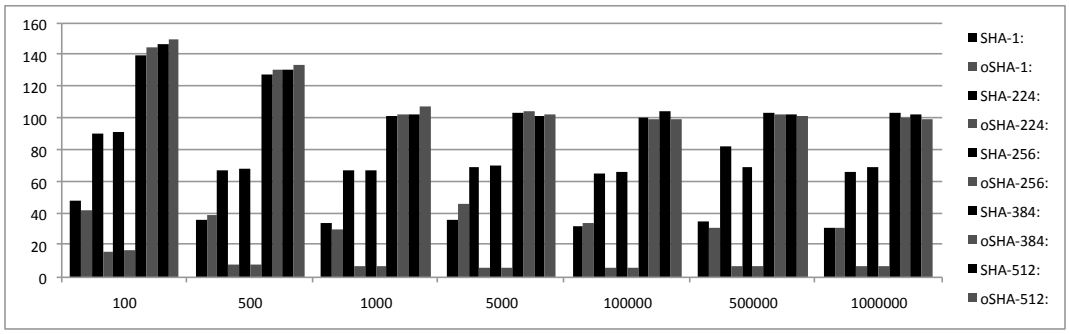


Figure A-1: Performance improvement after minor modification to SHA1/2 reference implementations

A.6 Summary

While security is certainly very important, for a hash function to be practical in some applications, it will also need to be efficient. This efficiency requirement has largely been overlooked in the literature. In this appendix, we not only show how to evaluate the efficiency of hash functions, we also discuss how to improve it. Although we present a set of optimisations techniques while considering Intel platforms, most of these are generic and apply to a wide range of platforms. Unlike most of the works in the literature, we here do not merely report on performance figures of a particular set of hash function implementations, we rather show how to optimise them. We demonstrated how seemingly minor optimisation may result in a great efficiency gain.